

/A CONCURRENCY METHOD/
AN IMPLEMENTATION ON A 3B2 NETWORK

by

JOHN E. MORRELL

B.M.E., Fort Hays State University, 1978

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

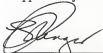
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by:



Major Professor

LD
2008
T4
1986
m67
c.2

A11206 737583

DEDICATION

For Denise, your love and support during these long three years enabled me to acheive a dream. For Troy, Ryan, and Trevor, knowing that you were missing Dad helped speed the work. Thank you for the prayers.

ACKNOWLEDGEMENTS

Thanks to Dr. Unger and Harvard Townsend for your patience with a "Band Director". I couldn't have finished this without your help. Thanks to the GTA's, you know who you are, for being a sounding board for ideas and plans.

Table of Contents

Table of Contents	ii
Table of Figures	iii
1. Introduction	1
2. Machine Architectures and Models for Concurrency	5
3. "A Concurrency Method" Concepts	16
4. System Environment	22
5. ACM Architecture and Implementation	29
6. Results and Conclusions	42
References	47
Appendix A — Sample Programs	50
Appendix B — ACM Program Code	53

Table of Figures

Figure 1. Single Instruction - Single Data (SISD) System	5
Figure 2. Multiple Processors With Global Bus	6
Figure 3. Single Instruction - Multiple Data (SIMD) System	9
Figure 4. Multiple Instruction - Multiple Data (MIMD) System	10
Figure 5. Hypercube Structure	11
Figure 6. Performance of Various Architecture Schemes	12
Figure 7. Examples of Petri Net Parallelism	13
Figure 8. Concurrency Modeled By Flow Graph	14
Figure 9. A Data Flow Program Graph	17
Figure 10. An ACM Program	19
Figure 11. A Portion of an Execution Tree	35
Figure 12. The Request Node Structure	37

1. Introduction

Most of the computers in use today utilize instruction sets which are based on the sequential processing techniques developed for early electronic digital computers. The rapid improvement of computer hardware has increased the speed of computation and the limits of accessible memory. However, these improvements have been based on the sequential processing techniques which have changed little, in comparison to changes in computer hardware, since UNIVAC I was a state-of-the-art computer.

Programming languages have evolved in a similar manner to computer hardware. Programming in low-level programming languages such as machine language or assembler has given way to programming in an easy-to-understand high-level programming language. The predominant high-level programming languages in use today are designed to be implemented on these sequential processing systems. Statements in a sequential program are executed one after another in a linear fashion. The computer programmer must take the responsibility to process the program's statements in the proper order so the results of the program will be correct and be in a form the user can understand.

Sequential processing does not take into account that some statements could be processed without regard of other statements. Statements must be processed sequentially in these systems because of primary memory access restrictions, as noted by von Neumann, and because there exists only one central processing unit (CPU) to handle statement execution [Killmon 1985]. This results in slower processing time if the CPU is required to share its resources with other processes as in a time-sharing system.

Redundant memory accesses may also slow down the processing of a sequential program. Most sequential languages force the system to do one calculation, store the result, do the next calculation, retrieve the first calculation, do some calculation with the first and second results, then store the result of this calculation and so on. This same process may be repeated over and over in a sequential program on a computer of standard architecture.

The desirability of the concurrent processing of programs can be noted in the development of virtual CPU or multiprogramming computers [Haber-man 1976]. Attempts have been made to make a sequential processing computer with one CPU appear to be a computer with several CPUs. Each user may be given a time-slice of an overall CPU cycle in which that user's program has access to the real CPU. The response time may be so quick that it may appear that the user has his own personal CPU. A system which supports this type of concurrency is still based on the same sequential processing techniques discussed previously. The response time in such a system can be slowed immensely if a large number of users are trying to access the CPU. As response time slows the appearance of a virtual CPU will decrease with a longer wait between acceptance of statements to be executed.

The modern "concurrent" programming language is usually based on some sequential programming language which is modified to give the appearance of concurrent processing. These modifications tend to make the language more complex than its sequential predecessor. The user must now be careful to ensure mutual exclusion in memory accesses during processing. These languages are still processed on sequential processing computers with virtual CPUs assigned to various "concurrent" processes. Each process must

wait for its time-slice before it gains access to the CPU and can be executed.

More and more tasks in today's society require quick processing of large and difficult operations. The need for methods and machines which allow true concurrent processing is great. Parallel processing would speed up processing operations allowing tasks to be completed more quickly. Concurrent processing could free the user from maintaining mutual exclusion in memory accesses and from specifying the sequence in which statements are to be executed. Many computer architectures and models have been put forth in an attempt to gain the advantages of concurrent processing and processing in distributed systems. Some of these architectures and models are discussed in Chapter 2. One of these models is A Concurrency Method (ACM) [Unger 1978].

Chapter 3 deals with the underlying concepts of ACM: data-driven processing, single-assignment of variables and data flow principles. Also discussed are the component parts of ACM: requests, actions and stimulating/terminating conditions.

Kansas State University does not have a computer designed for the type of concurrent processing needed for a desirable implementation of ACM. Such a computer would contain within one unit, multiple CPU's. Several machines of this type have been proposed but none is available yet commercially [Killmon 1985], [Kleinrock 1985], [Chang 1985], [Hindin 1985]. The AT&T 3B2-300 network of microcomputers networked together with an Ethernet interface provides the type of concurrent processing needed for this implementation. In addition, it is the environment in which an office information system might reside, thus providing a test environment to support other related work at the University. The physical characteristics of

vi
this hardware-software environment will be discussed in Chapter 4.

Chapter 5 deals with the architecture of ACM, the modules implemented and the structure of the request representation.

Finally, some results and conclusions from this effort are presented in Chapter 6.

2. Machine Architectures and Models for Concurrency

Research in computer architecture and design has led to several divergent paths of internal structures for the fast, efficient processing of data. Current studies in the area have developed such technologies as Reduced Instruction Set Computers (RISC) [Hindin 1985], Single Instruction stream - Multiple Data stream (SIMD) computers and Multiple Instruction stream - Multiple Data stream (MIMD) computers [Chang 1985] [Kleinrock 1985]. These designs are very different from the original computer architecture designed by von Neumann in the early days of computer development.

As stated earlier, most of the computers in use today are based on von Neumann's original concept of computer architecture. He proposed a single processing unit which accesses a single, one dimensional, sequentially addressed memory [Figure 1].

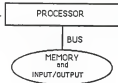


Figure 1. Single Instruction - Single Data (SISD) System

A program counter controls the flow of execution in a sequential program written in a sequential programming language. The processor accesses the memory by means of a simple bus which usually transfers data one word

at a time. This type of computer architecture is sometimes called a Single Instruction stream - Single Data stream (SISD) system. The users of such a system utilize programming languages such as FORTRAN, Pascal, BASIC and C which are best suited to the sequential operation of these machines. The user must be sure to enter the instructions in the correct order so the instructions will be executed in the proper order. When the processor is shared among several users, as in a time-sharing system, or among several processes, as in a "concurrent" programming language operating on an SISD system, a bottle-neck can occur if multiple users or processes attempt to access the same memory locations during their processing.

The ability to run processes concurrently would greatly speed execution of certain types of programs [Chang 1985]. A more recent architecture utilizes multiple processors with private memory resources which may access a shared global memory by means of a global bus [Figure 2].

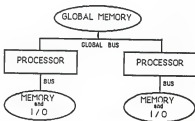


Figure 2. Multiple Processors With Global Bus

Brinch-Hansen [1978] and Hoare [1978] both proposed concurrent languages which would operate on a system which allows for parallel processing.

Both Brinch-Hansen's "Distributed Processes" (DP) and Hoare's "Communicating Sequential Processes" (CSP) chose the process as the fundamental unit of their proposed languages. In CSP a process P sends a message to process Q by executing a statement in the form

$Q \mid action(x) \{ \text{send the message to process Q} \}$

Process Q receives the message by including in the process body a command like

$P \mid action(y) \{ \text{get the message from process P} \}$

Both process Q and process P may execute their respective process statements in parallel until one of the processes tries to execute either the send or get statements in their process bodies. One process must wait for another process to be ready to receive or send a message. This means that one process may have to wait for the second process to reach its corresponding communication command. After the communication has been sent/received CSP allows both processes to continue execution of their respective process body statements. Ackerman (1982) and Chang (1985) point out that this type of computer architecture is not efficient if there is much interprocess communication required in execution of the jobstream since the global memory is available to only one processor at a time. Bottle-necks occur when several processes compete for access to the global memory. A hardware or software arbitrator must determine the priorities of the pending access requests, queue them and serve them sequentially, one request at a time. In this way, multiple processor systems with the global bus architecture become no different, functionally, than the single processor SISD system.

In implementations of "concurrent" languages on such systems the user must provide mutual exclusion of the global memory by use of semaphore operations or some other language construct which effectively locks out all other processes from accessing global memory. The creators of these "concurrent" languages have added concurrent constructs to essentially sequential programming languages. These additions must deal with the mutual exclusion problem and so add more constraints to the language which the user must deal with. This leads to increased complexity in the programming language. Clearly, this solution does not solve the need for an easily understood concurrent programming language.

Elimination of the global bus bottle-neck would minimize competition for memory and would reduce the amount of user-provided mutual exclusion. One way to solve the contention problem is to employ a channel technique. Many pieces of data are simultaneously processed by one operation command as in a vector processing system. Data is channeled from one processor to the next which sets up a serial chain of processing elements. Most super-computers and vector computers operate on the principle of a single instruction stream which operates on a multiple data stream (SIMD). They perform the same operation synchronously on different data. Vector processors operate on multiple-element vectors to speed completion of the task [Figure 3] [Chang 1985] [Killmon 1985] [Lackey 1985].

Using vector techniques, multiplication of two 100-element arrays requires only a single operation, not 100 sequential operations as in a scalar processor. Vector processors separate problems into scalar and vector portions. The vector portion of a problem is the "inner loop" of a problem. It defines the parts of the problem which are inherently parallel and may be

processed simultaneously. The scalar portion must be processed sequentially and this slows processing down [Killmon, 1985].

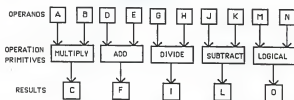


Figure 3. Single Instruction - Multiple Data (SIMD) System

Another way to resolve the problem is to surround the processing elements with additional register files. These buffers would allow operations to be pulled from memory and stored while operations are performed on them. The ability to have multiple processors with access to multiple resources naturally follows the SIMD architecture. An MIMD computer combines the benefits of the SIMD but also adds the power of multiple processors to the architecture [Figure 4] [Kleinrock 1985].

One MIMD machine is the IP - 1 developed by International Parallel Machine. Each processor has its own local memory, as well as parallel and serial I/O ports. The system utilizes multi-access memory (MAM) which allows all the processors to access memory at exactly the same time without arbitration. In this way, problems with a global bus bottle-neck are averted [Chang 1985].

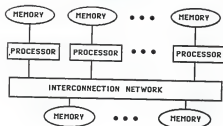


Figure 4. Multiple Instruction - Multiple Data (MIMD) System

Another parallel architecture computer is Intel Corporation's hypercube. The hypercube achieves concurrency through a collection of loosely coupled, independent processors which execute portions of a larger computational problem simultaneously [Figure 5] [Asbury 1985]. This large problem is broken down into partitions that can run independently on more than one processor.

The hypercube computer uses message passing rather than shared memory for communication among nodes of the networks. The process is the basic unit of cube computation which is performed by the message passing. A process is a sequential program (including system calls) which causes messages to be sent and received [Asbury, 1985]. Proponents of the hypercube architecture claim increased overhead efficiency compared to shared-memory schemes.

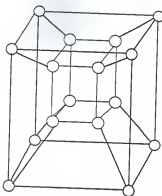


Figure 5. Hypercube Structure

A single node in the hypercube may contain multiple processes that perform computations. These computations can be distributed throughout the computer and executed concurrently. Each node is an autonomous computer system which has the capability of executing programs in a local mode if the user wishes to disengage from the system. The overall control of the hypercube is achieved by the "cube controller" which oversees the distribution of processes on the hypercube computer [Asbury 1985].

One process communicates with another process simply by opening up the channel and initiating send and receive requests. Message passing between nodes is asynchronous so there is no guarantee that a process has made a receive request before the node receives the message. To ensure message reception of messages, the node's operating system is used to buffer messages that are received prior to a receive request.

The hypercube computer makes use of a hybrid architecture in which some of the nodes are used as vector processors and some nodes are used as scalar processors. In this way partitions of the problem may be executed

extremely quickly with interprocess message passing between the vector and scalar nodes [Figure 6].

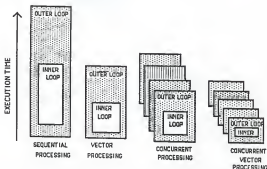


Figure 6. Performance of Various Architecture Schemes

The SIMD and MIMD architectures require a more sophisticated model for concurrency than the earlier SISD systems. CSP has had some impact on current theory but the languages built around the CSP concept slow computation on MIMD systems with its forced waiting during interprocess communication. This decrease in speed is not conducive to present needs for faster processing of data. Better models for concurrent programs and parallel processes needed to be developed.

One such model which was developed long before current MIMD architectures were developed is the Petri net [Peterson 1977]. A Petri net is a directed graph with two types of nodes - places and transitions. Places are drawn as circles and transitions are drawn as bars. These nodes are connected by directed arcs with the two types of nodes alternating. An input place is a place which leads to a transition and the output places are the

places immediately following the transition. A transition is enabled only if all of its input places contain a token or marker. The Petri net allows parallelism through the use of multiple tokens and by permitting more than one transition at a time to be enabled to fire [Figure 7].

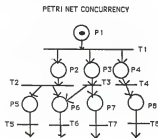


Figure 7. Examples of Petri Net Parallelism

An extension of the Petri net model was proposed by Drs. McBride and Unger of Kansas State University [McBride 1983]. They proposed a model for distributed systems based on Keller's work of Petri net models for a parallel program [Keller 1976] [McBride 1983].

The token is used to represent information concerning the execution of the job. A control net is developed in association with a token which describes the processing requirements of the job. The control net has the responsibility of directing the token's path through the distributed system. Information is shared among a subset of the processes in the system through the use of files that are explicitly shown in the procedure's Petri net.

Another method used for modeling concurrency on parallel processing systems is the control flow graph or the data flow graph. The data flow graph differs from the Petri net in the respect that the transitions are invisible to the user and are not represented on the graph. The nodes (places) are represented as circles and these nodes are connected by directed arcs. Process sequencing is determined by the placement of the node in the graph. High levels of concurrency are modeled in this way [Figure 8]. Each node might be considered as a "macro-graph" or a high level view of another flow graph. In this way, highly concurrent problems could be modeled in a top down method with more and more detailed refinement of each node.

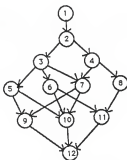


Figure 8. Concurrency Modeled By Flow Graph

Argument may be made that Petri nets and data flow graphs could be used to model concurrency on an SISD or global bus/shared memory system. However, since these systems do not permit concurrency or parallelism to the extent that SIMD and MIMD systems do, the Petri net based models can be used to exploit more modeled concurrency on the SIMD and

MIMD systems. The data flow graph can be used to model concurrency to a low level detail that coding a problem for execution on a parallel or concurrent processing machine is made very easy.

With the models and architectures becoming more readily available, algorithms and languages for these architectures are receiving more attention than when such technology was limited to individual research laboratories. The introduction of super-computer techniques for concurrent processing into cheaper super-minicomputers has allowed more people to be able to afford the speed and flexibility of the super-computer. The next chapter will describe a concurrent programming method which could be used on these data flow machines to provide an easy method for concurrent processing of complex problems.

3. "A Concurrency Method" Concepts

The architecture of "A Concurrency Method" (ACM) is based on two primary principles of data flow design, the single-assignment of variables and data-driven processes.

The concept of single-assignment of variables has its roots in mathematics. An example of a common programming practice is the statement $X = X + 1$. This expression has no meaning in a mathematical context since it implies that $0 = 1$ and other equally absurd statements. The name of the variable on the left hand side of the equal sign (=) stands for the memory location to which the result of the calculation on the right hand side of the expression is to be stored. In a case such as this the variable on the right hand side stands for the value which is presently at that memory location. The value at the present location is to be incremented and then stored in the same memory location. The left hand variable is usually changed to some unused variable name to distinguish between it and the operand variable. The expression could be written as $XX = X + 1$. This is the principle of single-assignment. Each time a new result is computed, that result is given a new name. When the material value associated with that variable name is no longer needed it is deleted from the system.

The concept of data-drive also comes from the field of mathematics. When a mathematical expression, such as $(A * B) + (C / D)$, is evaluated, only the portions of the equation whose values are known can be computed. In an example such as this either the expression $(A * B)$ or (C / D) can be processed if the values of their respective operands is known. Only after both sides of the plus sign (+) are evaluated can the addition take place.

Further, it makes no difference which of the addends is evaluated first. The sequencing of the computations is inherent in the problem. The concept of ordering instruction execution on the availability of required materials is known as data-driven processing.

The data flow graph is a directed graph which incorporates the concepts of data-drive and single-assignment. As discussed earlier, the data flow graph can be used to model a program of execution. Two types of nodes (places) are used to represent either operands or operators. The operands are often called tokens or data items, and are represented by circles. These tokens are the materials needed for inputs by operators or results obtained by operations on the input tokens. Operators are represented by boxes. Operators may be primitives which are embedded in the hardware or they may be more complex primitives built from combinations of the embedded primitives. The nodes are connected by means of directed arcs. The sample equation $X = (A * B) + (C / D)$ could be graphed as in Figure 9.

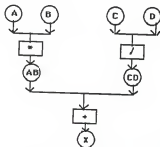


Figure 9. A Data Flow Program Graph

The results produced by the operations on the operands may be final or they may be passed on as operands to other operators which may then produce operands which are final results or be passed on to yet other operations. It should be noted that another concept of data flow processing is that of meaning associated with values in memory locations. In standard programming languages, no distinction is made as to whether this piece of data is an instruction or data to be used by the instruction. A characteristic of a programming language which uses multi-dimensional data structure is the need to break the concept of one variable being "next" to another in memory. This concept makes the understanding of SIMD and MIMD computers less arduous.

"A Concurrency Method" [Unger 1978] is the basis for this implementation. This language is naturally concurrent in that it incorporates the principles of data flow and single assignment, and using data-drive as a means of scheduling program execution. ACM allows data-drive to schedule processes which may run concurrently while blocking execution of those processes which rely on results of other processes as operands.

A user may make a "request" of ACM. The request is a statement that asks for an action to be performed. If the action is immediately executable it is said to be primitive. Otherwise the request is a set of requests which ask for more detailed actions to be performed. A simple action and a simple request are both represented by a three-tuple (m, a, r) : m is the material (operand) needed for execution, a is the action to be performed and r is the result of the action. Both m and r may be lists consisting of one or more data objects. A request in ACM must have all the required parts of the three-tuple. An example of such a request might be `SUBTRACT(Z;X,Y)`

where X and Y are the materials needed for execution (m), SUBTRACT is the name of the action to be performed (a), and Z is the result obtained from the action (r). ACM's data-driven scheduler would allow this action to execute only if the values of X and Y were available.

If the request is a list of requests or the action is a complex primitive, ACM allows the capability of refining or "detailing" an action or request into specific sets of requests until the requests at the lowest level are able to be executed by the host device's primitives. The process of detailing resembles the structured programming method of top-down design. This detailing process should be invoked automatically if an action is not immediately executable. The requests which are refinements of the parent request should be subjected to further refinement themselves if they are not primitives. Carter [1985] presents an example program to solve the quadratic equation for real roots. The action request is QUADROOT(R1,R2;A,B,C). After this request is made the refinement process would produce the detailing of Figure 10.

```

QUADROOT(R1,R2;A,B,C)
DIVIDE(R1; R11, A2)
DIVIDE(R2; R21, A2)
SUBTRACT(R11; NEGB, RAD)
ADD(R21; NEGB, RAD)
MULTIPLY(AC4; 4, AC)
SUBTRACT(RADSR; BB, AC4)
ROOT(RAD; RADSR)
SQUARE(BB; B)
MULTIPLY(AC; A, C)
MULTIPLY(A2; 2, A)
NEGATIVE(NEGB; B)

```

Figure 10. An ACM Program

ACM is strengthened by the addition of conditional statements which further control the sequencing of request execution. Stimulation and Termination conditions are Boolean conditions which allow the user or the system to determine if a request is valid and available for execution. Stimulation and Termination conditions are optional, but each request or action may have at most one stimulation and one termination condition. The conditions may also appear individually. The data-drive principle is augmented since the truth of the condition is a requirement for the request or action to be executed. If the stimulation condition is not met then the request or action will remain blocked. The termination condition controls the termination of a request or action. If the termination condition is true then the request is purged from the system, it is not eligible for execution. An example of a request with stimulation and termination is the statement

[Y <> 0] DIVIDE(Z;X,Y) [Y = 0]

To divide the value X by the value Y the stimulating condition of Y not being equal to 0 must be met. If Y is equal to 0 then the action would result in an error as division by 0 is undefined. Should this be the case, the request would be removed from consideration as an executable statement.

Conditionals may be entered as part of the request by the programmer when the program is created. This type of condition is called "external" as the programmer brought the condition from outside the program. If the programmer uses primitives which have been previously defined, these primitives may need to have conditions met of which the programmer may not be aware. In this situation the system will add the conditions to the primitives for the correct execution of the action. These conditions are said

to be "internal" in that the programmer does not need to know about them prior to program entry.

The concepts of ACM readily lend themselves to possible implementation on a SIMD or MIMD architecture. ACM provides an easy programming language in which the programmer does not need to worry about the order of execution unless he or she wishes. The automatic refinement process would allow a programmer to enter in a very high level program and have ACM do the top-down refinement which is sometimes tedious in programming languages such as Pascal and FORTRAN.

4. System Environment

After the decision to work on this project was made, one of the first questions which needed to be answered was "which network of computers available at Kansas State University offers the best system environment on which to do this implementation?" Further, this implementation would attempt to duplicate Carter's [1985] work at the University of Kansas. This final requirement suggested that the implementation be accomplished on a network of microcomputers.

The networks available at Kansas State included the PC Net of Zenith Z-150's with Ethernet as the network system, the Apple network of Macintosh personal computers using the Appletalk local area network communications software and hardware, and the newly acquired AT&T 3B2/300 super-microcomputers networked with AT&T's 3BNet network system.

The PC Net was quickly ruled out because the full Ethernet networking package was not implemented in the networks here. The ACM implementation would require the full Ethernet package being installed to allow user controllable message passing among all the nodes of the network. The current Z-150 network uses a Z-150 with a hard disk drive as a very limited "fileserv" from which nodes may access files. The files requested are then copied down to the node which requests them. Read only access for files on the fileserv is given to the nodes of the system. Message passing is limited to requests for file access from the nodes to the fileserv and the passing of those files from the fileserv to the nodes. This system would not allow the type of concurrency needed for the implementation of a naturally concurrent, data flow language.

The Kansas State University Department of Computer Science recently purchased several Apple MacIntosh personal computers with Apple's network Appletalk. This network of personal computers was judged to be preferable to the Z-150 network in that the entire network package was installed. This allows message passing among all the nodes of the network. The user interface is also very user friendly with the use of icons, pull-down menus, multiple windowing, and a mouse for easy cursor control. All of these features would make the use of the MacIntosh network for this implementation very nice. The only aspects of which were not favorable toward this network were: (1) the very large amount of application interface required for using the MacIntosh features; (2) the slow speed of the MacIntosh; and (3) that this network is not interfaced with the "switch" by which the computer department's distributed terminals could be used to access the MacIntosh network. For these reasons, the MacIntosh network was ruled out as the base "parallel processor system" for this implementation.

The AT&T 3B2/300 super-microcomputers seemed the best choice of all the microcomputer networks available. The system is new to Kansas State and is relatively unused, at this time, by courses offered by the computer science department. 3BNet was advertised to offer the type of message passing capabilities required for this implementation. The specific characteristics of 3BNet are presented later in this chapter. It was decided that the implementation of "A Concurrency Method" would be designed specifically for two or more AT&T 3B2/300 super-microcomputers connected with the 3BNet local area network, also from AT&T. The 3B2 network is accessible from any of the terminals on campus which are

connected to the "switch", a multiplexor which allows access to the department's various computing facilities. The user need not log on to the 3B2 consoles in Nichols hall in order to access the ACM implementation. All that is required is an account on the 3B2/300's.

The 3B2/300 super-microcomputer is a multiuser, desktop, 32-bit super-microcomputer that uses the AT&T WE-32000 microprocessor with a Random Access Memory (RAM) of 512 Kilobytes (user-expandable to 2 Megabytes). Increasing RAM memory greatly improves processor efficiency by providing a larger number of available buffers.

The terminal for the 3B2/300 system at Kansas State is the 4425 Dataspeed terminal from AT&T. This terminal is an asynchronous, serial, video display terminal (VDT). The terminal features a user-selectable 80- or 132-column screen. Five different character sets, full screen windowing, and a fully buffered auxiliary printer port are also standard features of the VDT. The keyboard is based on a standard QWERTY layout with several additional keys and features. It includes a 16-key numeric keypad, 38 downloadable function keys, and 8 cursor control keys. The function keys are dynamically labeled at the bottom of the display and can be used in applications programs designed for use with this terminal. Terminal emulation is also an important feature in this system as the Viewpoint terminals on the switch may also be used to access the 3B2 network. For this reason, this implementation of ACM will not utilize the function keys as the Viewpoint terminals do not have the function keys.

The actual 3B2/300 base unit contains the power supply, a cooling fan, and the WE-32000 microprocessor with its 512 Kilobytes to 2 Megabytes of Random Access Memory. The computer features two standard RS-232-C

serial ports with the capability to expand the Input/Output (I/O) ports so that peripherals may be added to the system. The system allows the user to install up to four feature cards with 4 serial ports and 1 parallel port per card. The total capacity is 18 serial and 4 parallel ports. The base unit also allows for the addition of the 3BNet feature card which is needed for connection to a 3BNet local area network.

The computer also features a 5.25 inch "floppy" disk drive, a battery powered, nonvolatile, time-of-day clock that retains time even when the power is turned off, and a nonvolatile RAM for saving essential information that is normally lost when power is removed or the user logs off of the system. The nonvolatile RAM feature also saves information which would be of great value to a system representative in the event of a system failure. A 10- or 30-Megabyte hard disk drive may also be added internally to the computer unit.

All of these components are tied together with AT&T's multiuser operating system, UNIX System V, Release 2.0. The concept behind the UNIX system is conducive to a multiuser system. The workbench concept is implemented through the users' ability to move from their home directories to other directories on the system, accessing files needed for software development, data sharing, and other needs. In this way, copies of infrequently used files do not need to be in each user's home directory, but may be kept in one accessible location, thereby saving storage space on the system. The user or "owner" of the file has control over the accessibility of his/her home directories and files. The access rights to a file or directory may be changed by the owner or by the super-user. This allows some measure of privacy for personal work while allowing access to files which

may be needed by other members of a programming team.

The UNIX operating system is written in the language, C. The user interface is the Bourne shell command line interpreter. The system also features several utilities packages which are useful in designing applications for the computer. These include: directory and file management command utilities, three different text editors, on-line help for UNIX commands and terms, data encryption capabilities, system administration utilities, screen handling routines, and utilities to enhance the user interface with the UNIX system. The 3B2 system at Kansas State currently supports the C and Pascal programming languages as well as the shell language used by the UNIX system. The implementation of Pascal on the 3B2 allows for calls to C programs, UNIX system commands, and FORTRAN routines. DEMON, DEbug MONitor, is also included. This utility allows the programmer to examine the monitor registers of the system.

Some optional programs and features available for the system are the BASIC and FORTRAN programming languages, Documenter's Workbench, Instructional Workbench, Writer's Workbench, Basic Networking, Graphics, Line Printer Spooling, Spell, and Terminal Filters.

The entire system of 3B2 computers and peripheral devices at KSU are tied together with the 3BNet local area network. The network claims to offer communication among nodes of the network without the delay of mail or relaying messages. This network can be used to tie together a combination of 3B2 computers and/or other machines equipped with Ethernet compatible hardware and software. The 3BNet package includes the feature card which is inserted in the 3B2 base unit. The feature card contains the hardware and firmware required for the 3BNet feature to interface with

other nodes on the local area network. These components include integrated circuits (IC's), memory IC's, resistors, and discrete units which are mounted to the card.

The package also includes a transceiver which provides the connection to the local networking cable. This is the connection to the common bus used for communication among the nodes of the network. A "drop" cable is included to connect the feature card in the computer base unit to the transceiver. A disk containing the software necessary for the network interface commands and programs is also included in the package.

The installation of 3BNet allows the user to transfer files and share information with those machines on the network. The user may also have the capability to execute commands among the nodes on the network using the UNIX system. 3BNet nodes are machines which use the UNIX operating system. They can be controlled by the user as long as the UNIX system is being used. The 3BNet hardware communicates with similar hardware attached to other computers on the network. The user enters a command and the software routine associated with that command manipulates the hardware to perform the operation. Through the connection to the network, the operation can be carried out on a remote machine. This feature is specifically what this implementation of ACM needed to allow the concurrent processing of requests among the various nodes of the network. In this way, the entire network of 3B2/300 super-microcomputers may be thought of as one large, multiple processor computer.

The application interface to 3BNet is provided by the open, close, read, write, and ioctl (i/o control) system calls of the UNIX operating system. Options specific to 3BNet are set with ioctl commands.

Access to the network is achieved through "virtual ports". Any process can open more than one virtual port. The number of ports that any one process can have open concurrently is limited by the number of NI (Network Interface) driver ports available and by the number number of files that a process can open. Ports opened by a process are inherited by any child process created by that process.

Packets transmitted from the 3B2 computer must be a multiple of 4 bytes. The NI driver will pad any packets which are not a multiple of 4 bytes long. The application interface must include a 14 byte header for each of its packets. This header is not from the packet by the NI driver.

When a packet sent to a virtual port arrives from another node, the packet is copied into the receive buffers for that part. Therefore the number and size of the receive buffers are set by the application. The NI driver will maintain a virtual port status table that may be obtained and modified by the application. Further technical information on the actual commands used to initiate this interface can be found in *AT&T 3BNet Manual [AT&T 1984]*.

5. ACM Architecture and Implementation

This implementation of "A Concurrency Method" is based on the work of Catherine Carter [1985] at the University of Kansas. The implementation was undertaken for three principle reasons: (1) to provide a working data flow simulation based on ACM concepts; (2) to allow a better understanding of concurrent processing, as related to distributed systems and parallel processing computers; and (3) to provide a central kernel of a data-flow processing operating system which could be used in other research at Kansas State University.

As has been discussed earlier, ACM is based on the principles that are the basis for a data flow computer. A data flow computer requires the same basic software components for operational control and user-machine interface as a computer of standard von Neumann architecture. The main controller of any computer system is the operating system. Since data-driven processing is an integral concept of ACM, the operating system must have a request scheduler which would determine if a request was ready for execution.

The single assignment aspect of ACM specifies that after the materials of a request are no longer required, the resources are deallocated and the request is removed from the system. For this reason the scheduler must also remove the request from the eligible request queue.

Another aspect of a data flow computer is the matter of concurrent execution. The operating system must determine if the concurrent execution is necessary and/or possible for waiting requests. It must also control and monitor the execution of these requests over the distributed system and

then return the results of the actions to the proper parent action. 3BNet was believed to be a possible means of implementing a data flow model. The literature on 3BNet suggested that the network allowed the sharing of data and the sending of messages from one node to other nodes of the network. If such claims proved true, the network could be controlled by the application program and effectively link all the WE-32000 microprocessors of the network into one parallel/multiprocessing machine. In this way 3BNet would be visualized as the interconnection bus between processors. Execution speed would also depend on the speed of the processors polling their message queues and the speed of transmission of the packets to other nodes. This implementation does not include any message passing capabilities. Other researchers may use this implementation as a core for various message passing schemes on the 3B2 Network.

A feature of data flow processing which was determined to be desirable was that of the even distribution of work among all the processors in the system. The operating system would need to determine the work load of all the processors in the system to ensure that some processors do not sit idle while other processors are overloaded past their capacity to accomplish the tasks requested of them. The system's load balancer would provide complete modularity to the system. Each processor would be autonomous and able to execute its requests without regard of the status of the other processors in the system. If only one processor of the system was working, the system would still function properly as a machine of standard von Neumann architecture. However, with only one processor, any attempt at concurrent execution is blocked.

Another aspect of this load balancing is that it occur automatically, without any user intervention, at periodic intervals. Once execution begins, the load balancer would determine which processor would execute which parts of the program. The scheduler, then, would determine which actions are executable and which requests would be blocked, waiting for the results of their refined requests. A dispatcher would then send the requests to the processors indicated by the load balancer.

An editor for program entry would also be needed for the user of the system to enter requests of the computer. Carter determined that a properly constructed editor would help speed up the translation process since her implementation translates each line of a program as it is entered. She also allowed the user to store and combine previously stored programs with programs being edited [Carter 1985, p. 27]. Such an editor could be used to limit the gap between the physical machine and program entry.

The user interface should be easy to understand and display messages in a way that will be "pleasing" to the user. Screen design, in this case, is a very important part of the user interface, since all messages about execution status of the requests from the user are displayed on the screen. The screen displays all the work currently on its associated microprocessor's request execution queue, as well as the current status of its user's program of requests.

Carter used the editor to work also as an interpreter. It was decided not to implement an interactive editor for this implementation so it was necessary to implement an interpreter which would parse the user's program requests to ensure the rules of proper ACM syntax are followed for each request. If a request was judged invalid the user would be notified, the

invalid request would be placed in a file for invalid requests, and the next request would be processed. The interpreter would also build the user's request tree as each successfully parsed request is processed.

The overall structure of some data flow machine implementations can be abstracted as being a tree of processors integrated with the software, thereby comprising the entire system. In this type of system there is no "root processor" to act as a master system controller so, in this sense, there is no real root to the tree structure. Each node of the system is equal to every other node of the system as far as the distribution of requests is concerned. The node on which the user is logged into the system becomes the "home node" for that user. This node name is used as the root of the request tree which is built in the home node environment of the user. Each valid user request is entered into the tree on a branch of the root. Each tree node in the first level of the tree represents a single request from the user's input file program. Each of these nodes then become the "root" of the tree to which the refinements of the request are added. As the refinements of the requests are further refined, if more refinement is needed, the second level of the tree also takes on the characteristics of the "root" of their tree of refinements. In this way the tree structure may begin to resemble a heap structure. The concept that each of these nodes represents a request leads to the treatment of each of these nodes of the tree structure as a single executable packet. Each packet, then, can move around the system as necessary for load balancing while maintaining the relationship of requests and their "parent" program within the user's home processor.

Operand values are be stored as tokens in this implementation. This scheme keeps the values of the operands as part of the request structure

instead of putting them in some other type of data structure and then matching requests for access to those values. Bottlenecks are avoided since there is no need to access shared memory to determine the value of an operand from this data structure. More memory is required for the request structure in this method but a separate process for matching operands and values is not needed.

As stated previously, it is be the interpreter's job to build and maintain the request tree for the user. After interpretation, the user's request tree is easily added the the execution tree of the home processor. As the request tree is built, the interpreter will refine each user request into its immediately executable primitive statements, adding these refinements into the "tree" of the parent request, then interpret the next request in a like manner, until the end of the input file is reached.

A major portion of the system which determines the "friendliness" of the user interface is the display of the various functions of the program on the screen. The display indicates the request currently being executed, the requests which are stimulated (ready) for execution, and the requests which are not yet ready for execution but are only requested. A label for each request moving through the distributed system is needed to be sure the results from the actions performed are returned to the correct home node. This also allows other users on the system to observe the requests being executed at their nodes and see the interaction of the distributed processing of the requests. It was also seen that some way to determine where a particular request fit into the tree hierarchy was also needed. It was decided that the user's login "account" name would be used for identification of the home node along with the letter of the terminal being used as the home

node. Since the 3B2/300 is a multiuser system it did not seem prudent to use just the name of the computer being used as the home node. It is possible that several users of the ACM system could be logged in to the same home node. Therefore, the combination of login name and home node name seemed to be an adequate label. The representation of the request's placement in the tree hierarchy was seen to be a simple matter of designating which level of the tree the request came from and the side of the "root" node. Each request from a specific user would contain a header of the user's name and the request's level in the tree. A program could then be reconstructed by putting all the user's requests in order according to their level designations. A typical request might be represented as

JOHNB 211 MULTIPLY(Z;X,Y),

where JOHN is the user's login name, B is the home node terminal (ksu3b2b), and 211 is the path of this request from the root of the execution tree. For this implementation the size of the user tree was limited to 9 descendents, numbered 1 through 9, from each node so that the displayed path would clearly identify the actual path of the node. A graphical representation of this example can be seen in Figure 11. With the above specifications taken into account and Carter's implementation as a guide, this implementation of "A Concurrency Method" was begun. C was chosen as the primary language. The UNIX operating system is written in C and offers many libraries of routines which would make programming this implementation much easier. In addition, it was determined that a C program will execute faster than another language on the UNIX system. The C language provides for the use of modular design, much like Pascal, and

allows for library programs as well as user defined programs to be "included" as part of the definition of the program unit.

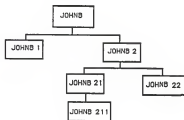


Figure 11. A Portion of an Execution Tree

It can be seen that there should be at least four modules to the ACM system: the scheduler, the tree manager, the network interface, and the display handler. Another module, decs.c, was created to centralize all of the global variables and data types required by the program. Also, it was decided to remove all the math functions from the RUN routine. All of these functions are in the file funcs.c. This allows for easier expansion of the available ACM language constructs in future work.

Following Carter's concept of the three types of execution tree nodes, each with the same structure, design of the C structure was begun. The user node specifies the current user's name and the work load required by this program tree. Interior nodes are used to hold requests which must be refined further and are used to pass the results from other actions to the proper consumers up and down the branch. The leaf nodes contain the exe-

cutable primitive operations which cannot be refined further.

Each structure can be seen as a record representing a separate, executable request. The state of each request, that is, whether or not the request has been stimulated and whether or not the request has been moved to the ready queue, is kept as a field within the structure itself. The path level of the node, whether the request is primitive or interior, the user's name, the home node address, and the name of the request (a combination of the user and home node names) are also kept within the request's structure to allow the operating system to move the request around on the system.

A model of the request node is seen in Figure 12.

TIME	HOME	PARENT
NAME		NODE TYPE
	PATH	
	ORIGINAL PATH	
MATERIALS	COMPLETE	USERNAME
RESULTS	ON READY LIST	NEXT READY NODE
PARENT NAME		COMMENTS
	NEXT	
STIMULATION		TERMINATION

Field	Purpose
TIME	Execution time required for this request
HOME	Address of this home node
PARENT	Pointer to the parent node (request)
NAME	Name of this Action or Request
NODE TYPE	Interior or leaf node indicator
PATH	Path from the root to this node
ORIGINAL PATH	Path in the home node
MATERIALS	Pointers to the materials nodes
RESULTS	Pointers to the results of this action
COMPLETE	TRUE if all materials are received
ON READY LIST	TRUE if request is stimulated
PARENT NAME	Name of the parent action or request
COMMENTS	Used for program (request) clarification
NEXT	Array of pointers' descendents
STIMULATION	Pointer to the stimulation node
TERMINATION	Pointer to the termination node

Figure 12. The Request Node Field Definitions

Because the primary structure is a tree, the program will be maintained in a tree format. Each request of the user's input program file is entered in the request tree. Refinement of each request is made immediately upon the determination of a valid ACM request. This refinement is displayed on the screen as indicated previously. When the end of the input file is reached and all refinement has been accomplished, the refined program is written to an output file in the tree format by use of a pre-order traversal of the tree.

The user is then prompted for the decision to execute the program or to exit the system.

The interpreter's operating system is required to manage the user's execution tree and the request tree. All nodes in the execution tree remain in the tree until the particular node is executed and/or its termination condition is met. In this implementation, the request tree is interleaved with the execution tree. As soon as a node from the request tree is ready to be executed, the state tag for that node is changed from "not ready" to "ready".

The CHKLOAD routine has the task of deciding how busy a particular node of the system is. To accomplish this task, the time required for execution of the primitives must be known. Carter [1985, pp. 39] assigned relative values to the times she discovered so that an overflow condition did not develop when computing the total workload of the processors.

There are several different communications schemes to be considered by future researchers using this implementation. Carter used a shared memory space among all the processors to ascertain the workload of each processor. The shared (global) memory is accessed by the load balancer in determining the loads of all the processors on the system. The load balancer first determines the current workload assigned to its processor and then requests access to the shared file which contains the workloads of the other processors. When access is granted by the shared memory semaphore, the balancer updates its entry to the current workload and then compares its load to the loads of the other processors in the system. If its load is greater than the average load at that point, it broadcasts a message to the other nodes requesting assistance with its load.

A processor may indicate its ability to accept more tasks to the system in general. In this case, the message handler searches the stimulated list of the processor(s) requesting assistance. The message handler selects requests from the overloaded processor's execution tree and sends them to the processor granting help.

Requests travel throughout the system and are stored in a node's message queue. It is important that incoming messages are received as soon as possible. For this reason, the routine which checks the message queue for new messages is called between all the other routines in the "shell" program. The shell program is the main program which calls all of the routines associated with ACM program execution. This method can cause a bottleneck by the use of the global memory. In large systems the possibility of more than one processor requesting access to the shared memory for updating its workload is increased.

Another method for communication between nodes would be to centralize the request distribution for the entire system into one node of the system. A predefined workload limit would be established for each node. After a node has reached its peak workload, further requests to that node would be sent to the message node for distribution to other nodes of the system. This message node would periodically poll the other nodes for their current workloads. This would allow the message node to decide which node would have the best chance of accepting the request. The path, station, and user name information in the request packet would allow the message node to send the results of the request to the proper user.

Again, this method could cause a bottleneck because all requests must pass through the central message node. If this node is busy, it is possible

that some requests may be lost by message buffer overflow. For this reason, it would be beneficial to have a timer of some sort which would reactivate a request from the send queue in the home processor. If the results are not received before the timer expires, the burden of processing the request would return to the home processor.

A general broadcast of the need for help in processing is yet another method of communication which could be used. When a processor meets the predefined workload limit a message is broadcast for help across the entire system along with the time needed for the next request. Other nodes would check their workloads and either grant or ignore the request for help. The request would then be sent to the node granting the call for help.

All three methods require the inclusion of some type of timeout device for handling the communication errors that might appear in the network. For this reason, requests are maintained in the home processor's tree until the results are returned. If the results are not returned within a certain time period, the home processor must take responsibility for executing the request.

An ACM program file must be created with one of the editors included as part of the 3B2 UNIX system with the file extension of *acm*.

The function which asks the user for the name of the program file looks for this extension. It is not necessary to type this extension in response to the prompt. If a "." is not found in the file name supplied by the user, the ".acm" extension is appended to the file name. If the file exists the program is then loaded into the parser and, if a valid ACM request, loaded into the execution tree.

Each request in the ACM program file must be preceded by its path number from the root. The path number is separated from the request by a ".". An example of some ACM program files is given in Appendix A. User refinement of predefined requests, such as QUADROOT, is not mandatory in the program file. If the request has a predefined refinement within the operating system, the refinement will be added to the tree with the program request as its parent. This relieves the programmer from the responsibility of refining some complex actions in the original program file. The completely parsed and refined ACM execution tree is then written to an output file in the form of *fn. out*, where *fn* is the file name supplied by the user in the sign on process.

Another file which is created by the system is *fn. res*, where, again, *fn* is the file name supplied by the user. This file contains all the results of the processing of the ACM program. Each line of the original file is written to the result file along with any messages about that request from the parser. Each screen displayed during the execution of the program is also written to the result file so that a comprehensive listing of all requests executed on that terminal are recorded in the order of execution. This provides the user with a complete record of his program's execution. However, a request which is processed on another node of the system will not appear in the output record of the user. It would be desirable to write any requests being sent to other nodes to the result file so the user can see where the requests from his program are being processed. This should be included in future implementations which utilize this system as a core kernel.

6. Results and Conclusions

This implementation of "A Concurrency Method" consists of several files which are included in the compilation of the main program, AMC.c. These files include: DECS.c, FUNCS.c, INIT.c, SCHEDULER.c, SCREENMGR.c, and TREEMGR.c.

DECS.c contains all of the constant definitions, structure definitions, and global variable declarations. The number of global variables is large due to the fact that C functions can only return one value. It was determined that several of the functions written for this implementation needed more than one value returned. This called for the use of "extern" or global variables.

FUNCS.c contains all of the ACM mathematical function definitions which are called by the EXECUTE function from SCHEDULER.c. It seemed that, by creating a separate file for the functions which are executable by this implementation, integration of more of the ACM language constructs would be made easier.

All of the routines which were seen as necessary to initialize the tree structure and all of the list structures are included in INIT.c. The SIGNON function initializes the user id for that particular terminal and sets the root of the execution tree.

SCREENMGR.c includes the functions which control most of the screen and file output for this implementation. These functions utilize the C library of "curses.h", which gives greater control over screen manipulation than the standard printf function. Curses.h was used primarily because the use of system call for such things as clearing the screen caused a bus error

and core dump.

The TREEMGR.c file contains the functions which initialize, build, and maintain the execution tree at the initial loading of an ACM program file. These functions determine which file is to be loaded into the tree and creates the result and output files. The result file contains all of the screens created during execution as well as a copy of the original input program with all of the messages appended by the parser. The final execution tree of the parsed ACM program is written to the output file. It should be possible to load and execute this file as an ACM program file.

C was an interesting language to learn. It appears that the implementation of C on the AT&T 3B2-300 computer does not support user defined types as well as some other implementations of C. Many type definitions were originally used to make the program easier to read. This allowed the C program to appear more like a Pascal program. Problems arose in the comparison of some of the user defined types. For this reason, all of the original type definitions were removed and all structures and variables were declared in the standard C style. The program is a little more difficult to read but there are no type conflicts during compilation.

This implementation has a bug in the PARSE routine. The first strcpy function encountered causes a bus error and core dump. Attempts to remedy this error were unsuccessful. The error has not been corrected due to lack of time to complete this report.

The implementation of this system has not been utilized in other research as of this date, so any results from the actual use of the system are based on research into the world of data flow processing and preconceived

ideas about this implementation in particular. It is believed that the use of ACM in an operating system course, a computer architecture course, or a programming language course would allow students to gain a better understanding of the characteristics and concepts of data flow processing and the architectures associated with these machines. It is further believed the the implementation of a message passing scheme which utilizes the system implemented in this research as a core would, on some level of granularity, provide a viable alternative to present concurrent programming languages.

It is expected that this implementation, when used with some sort of scheme to allow concurrent processing on other nodes within the 3B2 network, would execute slower than a comparable program written in some other high level language such as Pascal. There are two reasons for this belief. The first is, the Pascal language is written in such a way as to take advantage of the computer system's von Neumann architecture while ACM is designed to be used on a totally different type of architecture. Data transfer throughout the interior of a standard computer will be faster than data transfer throughout a network of computers which must observe certain communication protocols. The second reason is that the current implementation of the request structure requires much overhead as it is implemented with pointers. If the request structure could be implemented in a firmware or hardware memory structure then overhead would be reduced. It is believed, however, that at some level of granularity, even in this implementation, there would be some improvement in the speed of execution of the ACM language as opposed to a standard programming language such as Pascal.

Some improvements could be made through hardware and firmware implementations of various parts of the operating system. A separate processor could be used to implement the command interpreter which would increase the speed of command interpretation and also free other processors to do the work of executing requests. Firmware and hardware solutions could be used in implementing the request scheduler and tree handler, the load balancer, and message handler. Korosh Parazideh, graduate student at Kansas State University, is currently working on a possible hardware solution to a stimulation/termination processor.

Each processor in the system should have access to its own local memory. This would help to eliminate the bottleneck when multiple processors attempt to access shared memory. Multiple main memories are also needed, with the recommendation that all processors have access to all of the main memories. This could be accomplished by the implementation of channels from the processors to the memories.

It is believed that many of the constructs used in this implementation could be transferred from a software solution to a firmware/hardware implementation. This architecture design reduces hardware complexity while boosting performance in the parallel and concurrent execution of user requests. The optimization of the ACM operating system would also improve the speed of execution.

Even though the implementation is not finished, some tools and performance improvements can be seen at this time. One useful tool for the user's input of ACM programs would be an ACM syntax-sensitive editor. If this editor was constructed with a prescanner which flags all syntax errors, the interpreter would not have to be concerned with determining the validity of

the statements as far as ACM syntax is concerned.

A better interconnection network method is also needed. It appears that 3BNet is not as flexible as originally presumed. The 3BNet applications programmer is faced with a complex EtherNet applications interface for sending messages among nodes on the system. It might be possible to utilize the "nisead" commands for distributing requests among the nodes.

New algorithms for concurrent processing problems will need to be studied and tested. The introduction of data flow languages will also require a shift in the current concepts of computing. Students need to be exposed to the requirements of data flow processing and the new architectures to broaden their understanding of the concepts of data flow. This will prepare them for the time when data flow processing, and programming, will leave the research laboratory and enter the business and scientific computing world.

"A Concurrency Method" is an elegant language that is easy to understand once the basic concepts behind its design are learned. This language frees the programmer from the tedium of refining a high level algorithm into the proper primitive statements while trying to maintain the proper sequencing of the statements for proper execution.

"A Concurrency Method" will become an important tool as data flow computers become available to the world outside the research laboratory. It holds with the concepts of data driven processing with its adherence to the principles of data drive and single assignment. It will also be a valuable tool in the further study of data flow architecture and language design.

References

- . *AT&T 3B2/300 Computer Owner/Operator Manual*. AT&T Technologies, Inc., 1984, pp. 1/1 - E/2
- . *AT&T 3B2/300 Computer AT&T 3BNet Manual*. AT&T Technologies, Inc., 1984, pp. 1/1 - 8/44
- . *AT&T 3B2 Computer UNIX System V Release 2.0 Programmer Reference Manual*. AT&T Technologies, Inc., 1984, pp. 1/1 - 5/16
- Ackerman, William B. "Data Flow Languages", *Computer*, (February 1982), pp. 15 - 25.
- Asbury, Ray, Frison, Steven G., and Roth, Thomas. "Concurrent Computers Ideal For Inherently Parallel Problems", *Computer Design*, (September 1985), pp. 99 - 107.
- Carter, Catherine L. "A Concurrency Method On A Network Of Corvus Concept Personal Workstations", Master's Thesis, University of Kansas, 1985.
- Chang, Robin. "Parallel-Processing Computer Overcomes Memory Contention.", *Computer Design*, (September 1985), pp. 113 - 116.
- Gostelow, Kim P. and Thomas, Robert E. "A View of Data Flow", *National Computer Conference*, (1979), pp. 629 - 636.
- Gurd, John and Watson, Ian. "Data Driven System for High Speed Parallel Computing - Part 1: Structuring Software for Parallel Execution", *Computer Design*, (June 1980), pp. 91 - 100.
- Gurd, John and Watson, Ian. "Data Driven System for High Speed Parallel Computing - Part 2: Hardware Design", *Computer Design*, (July 1980), pp. 97 - 106.
- Haberman, A. N. *Introduction to Operating System Design*. Chicago, Illinois: Science Research Associates, Inc., 1976.
- Hansen, Per Brinch. *The Architecture of Concurrent Programs*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1977.
- Hindin, Harvey J. "Parallel Processing Promises Faster Program Execution", *Computer Design*, (August 1985), pp. 57 - 66.
- Hoare, C.A.R., "Communicating Sequential Processes", *Communications of the ACM*, Vol. 21, No. 8, 1978.
- Horowitz, Ellis. *Fundamentals of Programming Languages*. Rockville, Maryland: Computer Science Press, Inc., 1984.

Killmon, Peg. "Computers Tackle Challenges of the 90's", *Computer Design*, (December 1985), pp. 47 - 56.

Kleinrock, Leonard. "Distributed Systems", *Communications of the ACM*, Vol. 28 No. 11 (November 1985), pp. 1200 - 1213.

Kochan, Stephen G. *Programming in C*. Hasbrouck Heights, New Jersey: Hayden Book Co., 1983.

Laczko, Frank, and Demonico, Chris. "Parallel-Processing Building Blocks Lift 32-bit Performance", *Computer Design*, (February 1986), pp. 77 - 81.

Lackey, Stan, Veres, Jim, and Ziegler, Mike. "Supercomputer Expands Parallel Processing Options", *Computer Design*, (August 1985), pp. 76 - 81.

Lamport, Leslie. "The Mutual Exclusion Problem: Part 1 - A Theory of Interprocess Communication" *Journal of the ACM*, Vol. 33 No. 2 (April 1986), pp. 313 - 326.

McBride, Richard A. and Unger, Elizabeth A. "Modeling Jobs in a Distributed System", *ACM*, 1983, pp. 32 - 41.

Peterson, James L. "Petri Nets", *Computing Surveys*, Vol. 9 No. 3, (September 1977), pp. 223 - 250.

Plas, A. et al. "LAU System Architecture: A Parallel Data-Driven Processor Based on Single Assignment", *Proceedings of the Fifth Annual Symposium on Computer Architecture*. New York, ACM (1978), pp. 210 - 215.

Siewiorek, Daniel P., Anzelmo, Tony, and Moore, Russ. "Multiprocessor Computers Expand User Vistas.", *Computer Design*, (August 1985), pp. 70 - 75.

Sobell, Mark G. *A Practical Guide to the UNIX System*. Menlo Park, California: The Benjamin/Cummings Co., 1984.

Tanenbaum, Andrew S. *Computer Networks*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1981.

Treleaven, Philip C., Brownbridge, David R., and Hopkins, Richard P. "Data-Driven and Demand-Driven Computer Architecture", *Computing Surveys*, Vol. 14, No. 1 (March 1982), pp. 93 - 143.

Unger, Elizabeth A. "A Natural Model for Concurrent Computation", Ph.D. dissertation, University of Kansas, 1978.

Unger, Elizabeth A. and Schweppe, Earl J. "A Concurrent Model: Fundamentals", Paper presented at the 1978 MIT Workshop on Data Flow Systems and the 1979 Workshop on Data Flow at Toulouse with the 1st European Conference on Parallel Processing.

Wulff, Robert S. M. "Multiple Micros Distribute Text and Graphics Functions", *Computer Design*, (October 1984), pp. 141 - 147.

Appendix A
Sample Programs

Sample Program 1	51
Sample Program 2	52

Sample Program 1

This program shows the detailing of the QUADROOT function as it would appear after being parsed by the ACM program or as defined by the user before executing the ACM program.

```
1.QUADROOT(R1,R2:A,B,C)
11.Q1(AC,A2,NEGB,BB;A,B,C,2.0)
112.MULT(AC;A,C)
113.MULT(A2;A,2.0)
114.SUB(NEGB;0.0,B)
115.SQUARE(BB;B)
12.Q2(AC4;AC,4.0)
121.MULT(AC4;AC,4.0)
13.Q3(RADSR;BB,AC4)
131.SUB(RADSR;BB,AC4)
14.Q4(RAD;RADSR)
141.SQRT(RAD;RADSR)
15.Q5(R11,R12;NEGB,RAD)
151.SUB(R11;NEGB,RAD)
152.ADD(R12;NEGB,RAD)
16.Q6(R1,R2;R11,R12,A2)
161.DIVIDE(R1;R11,A2)
162.DIVIDE(R2;R12,A2)
```

Sample Program 2

This program shows the use of Stimulation and Termination conditions, the comment feature and the use of mixed case. The CONDENSE function is called before a program is parsed to eliminate blanks and change all characters to upper case.

```
1.[ x <> 0 ] DIVIDE(z; 4,2) [x = 0]
2.add (C; 2,3)!!These 3 statements have no connection
3.QuadRoOt(r1,r2;2.0,3.0,1.0)!!Note, no detailing with this one.
```

Appendix B

ACM Program Code

ACM Main Section	54
Global Declarations	56
Initialization Routines	59
ACM Math Functions	62
Scheduling Routines	69
Screen Management Routines	85
Tree Management Routines	90

A CONCURRENCY METHOD (ACM)

This section contains the main program, macro definitions, and include files. Also included are the functions FILERR and CONTROL.

```
/* C Library files to be included in the program */
#include <stdio.h>
#include <curses.h>
#include <string.h>
#include <math.h>
#include <ctype.h>

/* Library files to be included which were written by John Morrell 1986 */
#include "decs.c"
#include "init.c"
#include "screenmgr.c"
#include "treemgr.c"
#include "funcs.c"
#include "scheduler.c"

int FILERR()
{
    char ch;

    NEWSCREEN();
    noecho();
    mvprintw(10,1,"Do you wish to try again? (y/n)");
    refresh();
    ch = tolower(getch());
    addch(ch);
    refresh();
    while (ch != 'y' && ch != 'n')
    {
        mvprintw(18,1,"You must respond with a 'y' or an 'n'.");
        move(10,1);
        clrtoeol();
        mvprintw(10,1,"Do you wish to try again? (y/n)");
        refresh();
        ch = tolower(getch());
        addch(ch);
        refresh();
    }
    if (ch == 'y')
        return(TRUE);
    else
```

```

    return(FALSE);
} /* FILERR */

CONTROL()
{
    do
    {
        CHKLOAD();
        CHKQUIT();
        CHKBLOCKEDTREE();
        DISPATCH();
        CHKLOAD();
        MOVE();
        REFINE();
    }
    while (leop);
} /* CONTROL */

/* Begin main program of A Concurrency Method */

main ()
{
    INTRO();
    SIGNON();
    INITIALIZE();
    CHKQUIT();
    if (infile)
    {
        SHOWJOBS();
        CONTROL();
    }
    else
    {
        if (FILERR())
        {
            SHOWJOBS();
            CONTROL();
        }
    }
    QUITSCREEN();
} /* MAIN */

```

GLOBAL DECLARATIONS

This section contains all of the constant definitions, structure definitions and global declarations.

```
/* CONSTANTS */
```

```
#define MINREQ      1
#define MAXREQ      9
#define MAXLENGTH  80
#define S15         15
#define S30         30
#define S80         80
#define MAXNAME     25
#define MAXPATH     10
#define XCOORD      0
#define MINSTATION  1
#define MAXSTATION  2
```

```
/* STRUCTURE DEFINITIONS */
```

```
/* Conditional Structure */
```

```
struct cn
{ char   c_name[MAXNAME];
  char   c_cond[2];
  float  c_value;
  float  c_val;
  int    c_rcvd;
};
```

```
/* Materials Structure */
```

```
struct ms
{ char   m_name[MAXNAME];
  float  m_value;
  struct ms *m_next;
  int    m_rcvd;
};
```

```
/* Tree Structure */
```

```
struct tt
{ long   t_time;
  int    t_origpath[MAXPATH];
  int    t_station;
  int    t_path[MAXPATH];
  struct tt *t_parent;
  char   t_name[MAXNAME];
  struct ms *t_materials;
```

```

    struct ms *t_results;
    int      t_leaf;
    int      t_complete;
    char      t_uname[S15];
    int      t_onreadylist;
    struct tt *t_next1;
    struct tt *t_next[MAXREQ];
    char      t_prntname[MAXNAME];
    char      t_comments[S30];
    struct cn *t_stim;
    struct cn *t_term;
    int      t_done;
    struct tt *t_psend;
    char      t_what;
    int      t_where;
};

/* List Structure for Ready, Hold, and Sendlist */
struct lr
{
    struct tt *front;
    struct tt *rear;
};

/* GLOBAL VARIABLES */

int      eop;
int      ok;
int      toomuch;
FILE      *infile;
FILE      *tempfile;
FILE      *resultfile;
FILE      *outfile;
int      ycoord;
int      level;
int      load;
int      station;
int      screennum;
int      stimd;
int      termd;
struct lr ready;
struct lr hold;
struct lr sendlist;
int      path[MAXPATH];
char      user[S15];
char      fname[S15];
char      res[S15];
char      out[S15];
struct tt *tree;
struct tt *t;

```

```
struct tt *p;  
struct tt *q;  
struct tt *blocked;  
struct tt *now;  
WINDOW *curwin;  
WINDOW *readywin;  
WINDOW *blkwin;
```

INITIALIZE

This section contains the initialization routines necessary for the proper execution of the program.

TIMER(x)

```
int x;
{
    int y;

    x = x * 80000;
    for ( y = 0; y != x; ++y);
} /* TIMER */
```

NEWSCREEN()

```
{
    clear();
    refresh();
} /* NEWSCREEN */
```

SETRoot()

```
{
    int i;

    tree = (struct tt *)malloc(sizeof(struct tt));
    strcpy(tree->t_name, "ROOT");
    for (i = MINREQ; i <= MAXREQ; ++i)
        tree->t_next[i] = NULL;
    level = 0;
    tree->t_leaf = FALSE;
} /* SETROUTE */
```

SIGNON()

```
{
    char ch;
    int found, ready;

    NEWSCREEN();
    ready = FALSE;
    cuserid(user);
    myprintw(5,5,"Enter the letter of your 3b2 terminal (a,b,c,d,e) ");
    refresh();
    ch = tolower(getch());
    addch(ch);
    refresh();
    while (ch < 'a' || ch > 'e')
```

```

    {
        mvprintw(7,5,"%c is not a valid 3b2 terminal", ch);
        mvprintw(9,5,"Enter either a, b, c, d, or e");
        move(5,5);
        clrtoeol();
        mvprintw(5,5,"Enter the letter of your 3b2 terminal ");
        refresh();
        ch = tolower(getch());
        addch(ch);
        refresh();
    }
    strncat(user, &ch, 1);
    mvprintw(15,5,"Your user name is %s", user);
    refresh();
    SETROOT();
    TIMER(5);
} /* SIGNON */

```

RUNWINDOWS()

```

{
    curwin = newwin(3,COLS,2,0);
    readywin = newwin(10,COLS,5,0);
    blkwin = newwin(9,COLS,15,0);
} /* RUNWINDOWS */

```

INITIALIZE()

```

{
    int i;

    NEWSCREEN();
    mvprintw(10,0,"INITIALIZING . . . . .");
    refresh();
    RUNWINDOWS();
    screennum = 1;
    cop = FALSE;
    ready.front = (struct tt *) malloc (sizeof(struct tt));
    ready.rear = (struct tt *) malloc (sizeof(struct tt));
    hold.front = (struct tt *) malloc (sizeof(struct tt));
    hold.rear = (struct tt *) malloc (sizeof(struct tt));
    blocked = (struct tt *) malloc (sizeof(struct tt));
    for (i = MINREQ; i <= MAXREQ; ++i)
    {
        blocked->t_next[i] = (struct tt *) malloc (sizeof(struct tt));
        blocked->t_next[i] = NULL;
    }
    strcpy(blocked->t_name,"BLOCKED");
    blocked->t_leaf = FALSE;
    blocked->t_parent = (struct tt *) malloc (sizeof(struct tt));
}

```

```
blocked->t_time = 0;
sendlist.front = (struct tt *) malloc (sizeof(struct tt));
sendlist.rear = (struct tt *) malloc (sizeof(struct tt));
now = (struct tt *) malloc (sizeof(struct tt));
TIMER(3);
} /* INITIALIZE */
```

ACM MATH FUNCTIONS DEFINITIONS

This section contains all of the predefined math functions included in this implementation.

int ODD(x)

```
{
    int i;
    int b;

    b = 1;
    for (i = 1; i <= x; ++i)
        b *= -1;
    return(b);
} /* ODD */
```

ADD()

```
{
    struct ms *m;
    struct ms *n;

    m = now->t_results;
    m->m_rcvd = TRUE;
    m->m_value = 0;
    n = now->t_materials;
    while (n != NULL)
    {
        m->m_value += n->m_value;
        n = n->m_next;
    }
    n = now->t_results->m_next;
    while (n != NULL)
    {
        n->m_value = now->t_results->m_value;
        n->m_rcvd = TRUE;
        n = n->m_next;
    }
} /* ADD */
```

SUB()

```
{
    struct ms *m;
    struct ms *n;

    m = now->t_results;
    m->m_rcvd = TRUE;
```

```

n = now->t_materials;
m->m_value = n->m_value;
n = n->m_next;
while (n != NULL)
{
    m->m_value = n->m_value;
    n = n->m_next;
}
n = m->m_next;
while (n != NULL)
{
    n->m_value = m->m_value;
    n->m_rcvd = TRUE;
    n = n->m_next;
}
} /* SUB */

```

```

ADDMATS()
{
    struct ms *m;
    struct ms *n;

    m = now->t_materials;
    n = now->t_results;
    while (m != NULL)
    {
        n->m_value = m->m_value;
        n->m_rcvd = TRUE;
        m = m->m_next;
        n = n->m_next;
    }
} /* ADDMATS */

```

```

MULTIPLY()
{
    struct ms *m;
    struct ms *n;

    m = now->t_results;
    m->m_rcvd = TRUE;
    m->m_value = 1;
    n = now->t_materials;
    while (n != NULL)
    {
        m->m_value *= n->m_value;
        n = n->m_next;
    }
}

```

```

n = m->m_next;
while (n != NULL)
{
    n->m_value = m->m_value;
    n->m_rcvd = TRUE;
    n = n->m_next;
} /* MULTIPLY */

```

```

DIVIDE()
{
    struct ms *m;

    now->t_results->m_rcvd = TRUE;
    if (now->t_materials->m_next->m_value != 0)
        now->t_results->m_value =
            now->t_materials->m_value / now->t_materials->m_next->m_value;
    else
        now->t_results->m_value = 0;
    m = now->t_results->m_next;
    while (m != NULL)
    {
        m->m_value = now->t_results->m_value;
        m->m_rcvd = TRUE;
        m = m->m_next;
    }
} /* DIVIDE */

```

```

POWER()
{
    struct ms *m;
    int neg;
    int i;
    float a;
    float b;
    double c;

    neg = FALSE;
    a = now->t_materials->m_value;
    b = now->t_materials->m_next->m_value;
    if (a < 0)
    {
        a = abs(a);
        if (ODD(b) < 0)
            neg = TRUE;
    }
    c = exp(b * log(a));
}

```

```

now->t_results->m_rcvd = TRUE;
if (neg)
  now->t_results->m_value = -c;
else
  now->t_results->m_value = c;
m = now->t_results->m_next;
while (m != NULL)
{
  m->m_value = now->t_results->m_value;
  m->m_rcvd = TRUE;
  m = m->m_next;
}
} /* POWER */

```

```

SQRT()
{
  struct ms *m;
  struct ms *n;

  m = now->t_materials;
  n = now->t_results;
  if (m->m_value <= 0)
    n->m_value = 0;
  else
  {
    n->m_value = sqrt(m->m_value);
  }
  n->m_rcvd = TRUE;
}

```

```

SQUARE()
{
  struct ms *m;

  now->t_results->m_value =
    now->t_materials->m_value * now->t_materials->m_value;
  now->t_results->m_rcvd = TRUE;
  m = now->t_results->m_next;
  while (m != NULL)
  {
    m->m_value = now->t_results->m_value;
    m->m_rcvd = TRUE;
    m = m->m_next;
  }
} /* SQUARE */

```

```

SINE()
{

```

```

struct ms *m;
struct ms *n;

m = now->t_materials;
n = now->t_results;
n->m_value = sin(m->m_value);
n->m_rcvd = TRUE;
m = n->m_next;
while (m != NULL)
{
    m->m_value = n->m_value;
    m->m_rcvd = TRUE;
    m = m->m_next;
}
} /* SINE */

```

```

COS()
{
    struct ms *m;
    struct ms *n;

    m = now->t_materials;
    n = now->t_results;
    n->m_value = cos(m->m_value);
    n->m_rcvd = TRUE;
    m = n->m_next;
    while (m != NULL)
    {
        m->m_value = n->m_value;
        m->m_rcvd = TRUE;
        m = m->m_next;
    }
} /* COS */

```

```

TANO
{
    struct ms *m;
    struct ms *n;

    m = now->t_materials;
    n = now->t_results;
    n->m_value = tan(m->m_value);
    n->m_rcvd = TRUE;
    m = n->m_next;
    while (m != NULL)
    {
        m->m_value = n->m_value;
        m->m_rcvd = TRUE;
    }
}

```

```

        m = m->m_next;
    } /* TAN */

```

```

NL()
{
    struct ms *m;
    struct ms *n;

    m = now->t_materials;
    n = now->t_results;
    n->m_value = log(m->m_value);
    n->m_rcvd = TRUE;
    m = n->m_next;
    while (m != NULL)
    {
        m->m_value = n->m_value;
        m->m_rcvd = TRUE;
        m = m->m_next;
    }
} /* NL */

```

```

EXPO()
{
    struct ms *m;
    struct ms *n;

    m = now->t_materials;
    n = now->t_results;
    n->m_value = exp(m->m_value);
    n->m_rcvd = TRUE;
    m = n->m_next;
    while (m != NULL)
    {
        m->m_value = n->m_value;
        m->m_rcvd = TRUE;
        m = m->m_next;
    }
} /* EXP */

```

```

IROOT()
{
    struct ms *m;
    struct ms *n;

    m = now->t_materials;
    n = now->t_results;

```

```

if (m->m_value < 0)
{
    n->m_value = 0;
    n->m_rcvd = TRUE;
    n->m_next->m_value = SQRT(abs(m->m_value));
    n->m_next->m_rcvd = TRUE;
}
else
{
    n->m_value = SQRT(m->m_value);
    n->m_rcvd = TRUE;
    n->m_next->m_value = 0;
    n->m_next->m_rcvd = TRUE;
}
} /* IROOT */

```

SCHEDULING ROUTINES

This section contains all of the routines needed for determining when a request may be serviced.

DOSENDLIST()

```
{
    if (sendlist.front == NULL)
    {
        sendlist.front = p;
        p->t_psend = NULL;
        sendlist.rear = p;
    }
    else
    {
        sendlist.rear->t_psend = p;
        sendlist.rear = p;
        p->t_psend = NULL;
    }
} /* DOSENDLIST */
```

REDO()

```
{
    if (hold.front == p)
    {
        hold.front = p->t_next1;
        if (hold.front == NULL)
            hold.rear = NULL;
        q = hold.front;
    }
    else
    {
        q = hold.front;
        while (q->t_next1 != p)
            q = q->t_next1;
        q->t_next1 = p->t_next1;
        if (q->t_next1 == NULL)
            ready.rear = q;
    }
    p->t_next1 = NULL;
} /* REDO */
```

MOVE()

```
{
    struct tt *p, *q;
    int i, j;
```

```

int done;

p = hold.front;
while (p != NULL)
{
    if (p->t_path[1] == 0)
    {
        i = MINREQ;
        done = FALSE;
        while (!done)
            if (i > MAXREQ)
                done = TRUE;
            else
                if (blocked->t_next[i] == NULL)
                    done = TRUE;
                else
                    ++i;
            if (i <= MAXREQ)
            {
                p->t_path[1] = i;
                PUTINTREE(p, blocked);
                REDO();
                p = q->t_next1;
            }
            else
                p = NULL;
        }
        else
            p = p->t_next1;
    }
    SHOWJOBS();
    TIMER(3);
} /* MOVE */

```

```

REFINE()
{
    struct tt *p, *q;
    int i, j;
    struct ms *l, *m;
    struct cn *c;

    p = hold.front;
    while (p != NULL)
    {
        j = 0;
        for (i = 0; i <= MAXPATH; ++i)
            if (p->t_path[i] != 0)
                ++j;
        if (j > 1)
        {

```

```

if (p->t_leaf)
{
    q = NODE(i-2, blocked, p->t_path);
    if (q != NULL)
    {
        PUTINTREE(p, blocked);
        REDO();
    }
    else
        p = p->t_next1;
}
else
{
    p->t_complete = TRUE;
    m = p->t_materials;
    while (m != NULL)
        if (m->m_rcvd)
        {
            p->t_complete = FALSE;
            m = NULL;
        }
        else
            m = m->m_next;
    if (p->t_complete)
    {
        if (p->t_stim != NULL)
        {
            c = p->t_stim;
            m = p->t_materials;
            while (m != NULL)
                if (c->c_name == m->m_name)
                {
                    c->c_val = m->m_value;
                    c->c_rcvd = TRUE;
                    m = NULL;
                }
                else
                    m = m->m_next;
        }
        if (p->t_term != NULL)
        {
            c = p->t_term;
            m = p->t_materials;
            while (m != NULL)
                if (c->c_name == m->m_name)
                {
                    c->c_val = m->m_value;
                    c->c_rcvd = TRUE;
                    m = NULL;
                }
                else

```

```

        m = m->m_next;
    }
    PUTINTREE(p, blocked);
    REDO();
    p = q;
}
else
    p = p->t_next1;
}
}
}
SHOWJOBS();
TIMER(3);
} /* REFINE */

```

```

SENDON(p)
struct tt *p;
{
    struct ms *m;
    struct ms *n;
    int i;

    if (p != NULL)
    {
        m = q->t_results;
        while (m != NULL)
        {
            n = p->t_materials;
            while (n != NULL)
            {
                if (m->m_name == n->m_name)
                {
                    if (ln->m_rcvd)
                    {
                        n->m_value = m->m_value;
                        n->m_rcvd = TRUE;
                        n = n->m_next;
                    }
                    else
                        n = n->m_next;
                }
                else
                {
                    n = n->m_next;
                    m = m->m_next;
                }
            }
        }
        if (p->t_stim != NULL)
        {
            m = q->t_results;
            while (m != NULL)
            {
                if (m->m_name == p->t_stim->c_name)
                {
                    p->t_stim->c_val = m->m_value;
                    p->t_stim->c_rcvd = TRUE;
                }
            }
        }
    }
}

```

```

        } m = NULL;
    }
    else
        m = m->m_next;
}
if (q->t_term != NULL)
{
    m = p->t_results;
    while (m != NULL)
    {
        if (m->m_name == q->t_term->c_name)
        {
            q->t_term->c_val = m->m_value;
            q->t_term->c_rcvd = TRUE;
            m = NULL;
        }
        else
            m = m->m_next;
    }
    if (!p->t_leaf)
        for (i = MINREQ; i <= MAXREQ; ++i)
            SENDON(p->t_next[i]);
}
} /* SENDON */

```

```

PASSDOWN(q)
struct tt *q;
{
    struct tt *p;
    struct ms *m;
    struct ms *n;

    p = blocked->t_next[q->t_path[0]];
    SENDON(p);
    p = hold.front;
    while (p != NULL)
    {
        m = q->t_results;
        while (m != NULL)
        {
            n = p->t_materials;
            while (n != NULL)
            {
                if (m->m_name == n->m_name)
                {
                    if (!n->m_rcvd)
                    {
                        n->m_value = m->m_value;
                        n->m_rcvd = TRUE;
                        n = n->m_next;
                    }
                }
                else

```

```

        n = n->m_next;
    else
        n = n->m_next;
        m = m->m_next;
    }
    if (p->t_stim != NULL)
    {
        m = q->t_results;
        while (m != NULL)
            if (m->m_name == p->t_stim->c_name)
            {
                p->t_stim->c_val = m->m_value;
                p->t_stim->c_rcvd = TRUE;
                m = NULL;
            }
            else
                m = m->m_next;
    }
    if (p->t_term != NULL)
    {
        m = q->t_results;
        while (m != NULL)
            if (m->m_name == p->t_term->c_name)
            {
                p->t_term->c_val = m->m_value;
                p->t_term->c_rcvd = TRUE;
                m = NULL;
            }
            else
                m = m->m_next;
    }
    p = p->t_next1;
} /* PASSDOWN */

```

PASSUP()

```

{
    struct ms *m;
    struct ms *n;
    struct cn *c;

    if (now->t_parent->t_name == "BLOCKED")
        SHOWJOBS();
    else
    {
        m = now->t_results;
        while (m != NULL)
        {
            n = now->t_parent->t_results;
            while (n != NULL)

```

```

        if (m->m_name == n->m_name)
        {
            n->m_value = m->m_value;
            n->m_rcvd = TRUE;
            n = NULL;
        }
        else
            n = n->m_next;
        m = m->m_next;
    }
    m = now->t_results;
    while (m != NULL)
    {
        n = now->t_parent->t_materials;
        while (n != NULL)
        {
            if (m->m_name == n->m_name)
            {
                n->m_value = m->m_value;
                n->m_rcvd = TRUE;
                n = NULL;
            }
            else
                n = n->m_next;
        }
        m = m->m_next;
    }
    if (now->t_parent->t_stim != NULL)
    {
        c = now->t_parent->t_stim;
        m = now->t_results;
        while (m != NULL)
        {
            if (m->m_name == c->c_name)
            {
                c->c_val = m->m_value;
                c->c_rcvd = TRUE;
                m = NULL;
            }
            else
                m = m->m_next;
        }
    }
    if (now->t_parent->t_term != NULL)
    {
        c = now->t_parent->t_term;
        m = now->t_results;
        while (m != NULL)
        {
            if (m->m_name == c->c_name)
            {
                c->c_val = m->m_value;
                c->c_rcvd = TRUE;
                m = NULL;
            }
            else

```

```

        m = m->m_next;
    }
} /* PASSUP */

```

EXECUTE()

```

{
    struct ms *l;
    struct ms *m;
    struct ms *n;
    int i;
    int j;
    int done;
    struct tt *p;

    SHOWJOBS();
    TIMER(3);
    if (now->t_leaf)
    {
        if (now->t_name == "ADD")
            ADD;
        else if (now->t_name == "SUB")
            SUB;
        else if (now->t_name == "ADDMATS")
            ADDMATS;
        else if (now->t_name == "MULTIPLY")
            MULTIPLY;
        else if (now->t_name == "DIVIDE")
            DIVIDE;
        else if (now->t_name == "POWER")
            POWER;
        else if (now->t_name == "SQRT")
            SQRT;
        else if (now->t_name == "SQUARE")
            SQUARE;
        else if (now->t_name == "SINE")
            SINE;
        else if (now->t_name == "COS")
            COS;
        else if (now->t_name == "TAN")
            TAN;
        else if (now->t_name == "NL")
            NL;
        else if (now->t_name == "EXP")
            EXP;
        else if (now->t_name == "IROOT")
            IROOT;

        SHOWJOBS();
    }
}

```

```

TIMER(3);
PASSUP();
if (now->t_done)
{
    i = MINREQ;
    while (now->t_parent->t_next[i] != now)
        ++i;
    now->t_parent->t_next[i] = NULL;
    free(now);
}
else
{
    now->t_onreadylist = FALSE;
    m = now->t_results;
    while (m != NULL)
        m->m_rcvd = FALSE;
}
now = NULL;
SHOWJOBS();
TIMER(3);
} /* end IF LEAF */
else
{
    PASSUP();
    PASSDOWN(now);
    if (now->t_done)
    {
        i = MINREQ;
        while (now->t_parent->t_next[i] != now)
            ++i;
        now->t_parent->t_next[i] = NULL;
        for (i = MINREQ; i <= MAXREQ; ++i)
        {
            if (now->t_next[i] != NULL)
            {
                j = MINREQ;
                done = FALSE;
                while (!done)
                {
                    if (j > MAXREQ)
                        done = TRUE;
                    else
                        if (now->t_parent->t_next[j] == NULL)
                            done = TRUE;
                        else
                            ++j;
                }
                if (j < MAXREQ)
                    now->t_parent->t_next[j] = now->t_next[i];
                else
                    WAIT(now->t_next[i]);
            }
        }
    }
}

```

```

        free(now);
    } /* end IF DONE */
else
{
    now->t_onreadylist = FALSE;
    m = now->t_results;
    while (m != NULL)
        m->m_rcvd = FALSE;
    }
    now = NULL;
    SHOWJOBS();
}
} /* EXECUTE */

```

```

DISPATCH()
{
    if (ready.front != NULL)
    {
        now = ready.front;
        ready.front = ready.front->t_next1;
        if (ready.front == NULL)
            ready.rear = NULL;
        now->t_next1 = NULL;
        EXECUTE();
    }
} /* DISPATCH */

```

```

PUTON(q)
{
    struct tt *q;
    {
        if (q->t_complete)
            if (q->t_onreadylist)
            {
                if (ready.front == NULL)
                {
                    q->t_onreadylist = TRUE;
                    ready.front = q;
                    ready.rear = q;
                }
            }
            else
            {
                q->t_onreadylist = TRUE;
                ready.rear->t_next1 = q;
                ready.rear = q;
            }
    }
} /* PUTON */

```

```

SEARCH(p)
struct tt *p;
{
    int i;

    for (i = MINREQ; i <= MAXREQ; ++i)
        if (p->t_next[i] != NULL)
        {
            PUTON(p->t_next[i]);
            SEARCH(p->t_next[i]);
        }
} /* SEARCH */

```

```

BLOCKToready()
{
    SEARCH(blocked);
} /* BLOCKToready */

```

```

DELETEBRANCH(p)
struct tt *p;
{
    struct tt *q;
    int i;

    q = p->t_parent;
    i = MINREQ;
    while (q->t_next[i] != p)
        ++i;
    p->t_parent = NULL;
    q->t_next[i] = NULL;
    return(p);
} /* DELETEBRANCH */

```

```

int SETCOND(c)
struct cn *c;
{
    int state;

    if (lc->c_rcvd)
        state = FALSE;
    else
    {
        state = FALSE;
        if (c->c_cond == "<")
            if (c->c_val < c->c_value)

```

```

        state = TRUE;
    else if (c->c_cond == "<=")
        if (c->c_val <= c->c_value)
            state = TRUE;
    else if (c->c_cond == "=")
        if (c->c_val == c->c_value)
            state = TRUE;
    else if (c->c_cond == ">")
        if (c->c_val > c->c_value)
            state = TRUE;
    else if (c->c_cond == ">=")
        if (c->c_val >= c->c_value)
            state = TRUE;
    else if (c->c_cond == "!=")
        if (c->c_val != c->c_value)
            state = TRUE;
    }
    return(state);
} /* SETCOND */

```

CHECKCOND()

```

struct tt *p;
{
    int ready;
    int over;

    if (p->t_stim == NULL && p->t_term == NULL)
    {
        stimd = TRUE;
        termd = FALSE;
        p->t_done = TRUE;
    }
    if (p->t_stim != NULL && p->t_term == NULL)
    {
        ready = SETCOND(p->t_stim);
        if (ready)
        {
            stimd = TRUE;
            termd = FALSE;
            p->t_done = TRUE;
        }
        else
        {
            stimd = FALSE;
            termd = FALSE;
            p->t_done = FALSE;
        }
    }
    if (p->t_stim == NULL && p->t_term != NULL)
    {

```

```

over = SETCOND(p->t_term);
if (over)
{
    stimd = TRUE;
    termd = TRUE;
    p->t_done = TRUE;
}
else
{
    stimd = TRUE;
    termd = FALSE;
    p->t_done = FALSE;
}
}
if (p->t_stim != NULL && p->t_term != NULL)
{
    ready = SETCOND(p->t_stim);
    over = SETCOND(p->t_term);
    if (ready && over)
    {
        stimd = TRUE;
        termd = TRUE;
        p->t_done = TRUE;
    }
    else if (ready && !over)
    {
        stimd = TRUE;
        termd = FALSE;
        p->t_done = TRUE;
    }
    else if (!ready && over)
    {
        stimd = FALSE;
        termd = TRUE;
        p->t_done = TRUE;
    }
    else if (!ready && !over)
    {
        stimd = FALSE;
        termd = FALSE;
        p->t_done = FALSE;
    }
}
} /* CHECKCOND */

```

```

CHECKLIST(m)
struct ms *m;
struct tt *p;
{
    p->t_complete = TRUE;
}

```

```

while (m != NULL)
{
    if (lm->m_rcvd)
        p->t_complete = FALSE;
    m = m->m_next;
}
} /* CHECKLIST */

```

```

CHECK(p)
struct tt *p;
{
    int i;
    int j;
    struct ms *l;
    struct ms *m;
    struct tt *q;

    if (p != NULL)
    {
        if (p->t_name != "BLOCKED")
        {
            CHECKCOND();
            if (stimd)
            {
                if (!p->t_complete)
                    CHECKLIST(p->t_materials);
                if (p->t_complete)
                    if (!p->t_leaf)
                        CHECKLIST(p->t_results);
            }
        }
        if (!p->t_leaf)
            for (i = MINREQ; i <= MAXREQ; ++i)
                CHECK(p->t_next[i]);
    }
    return(p);
} /* CHECK */

```

```

CHKBLOCKEDTREE()
{
    CHECK(blocked);
    BLOCKTOREADY();
    SHOWJOBS();
    TIMER(3);
} /* CHKBLOCKEDTREE */

```

```

CHKTREE(p)

```

```

struct tt *p;
{
    int i;

    if (p != NULL)
        for (i = MINREQ; i <= MAXREQ; ++i)
        {
            CHKTREE(p->t_next[i]);
            if (p->t_parent != NULL)
                p->t_parent->t_time += p->t_time;
        }
} /* CHKTREE */

CHKWAIT(p)
struct tt *p;
{
    while (p != NULL)
    {
        if (p->t_leaf)
            load += p->t_time;
        p = p->t_next[i];
    }
} /* CHKWAIT */

INITTREE(p)
struct tt *p;
{
    int i;

    if (p != NULL)
        if (!p->t_leaf)
        {
            p->t_time = 0;
            for (i = MINREQ; i <= MAXREQ; ++i)
                INITTREE(p->t_next[i]);
        }
    return(p);
} /* INITTREE */

CHKLOAD()
{
    INITTREE(blocked);
    CHKTREE(blocked);
    load = blocked->t_time;
    CHKWAIT(hold.front);
} /* CHECKLOAD */

```

```
RUN0
```

```
{  
    LOADFILE();  
    SHOWJOBS();  
    TIMER(3);  
} /* RUN */
```

```
CHKQUIT0
```

```
{  
    char ch;  
  
    echo();  
    eop = FALSE;  
    if (tree->t_next[0] == NULL)  
    {  
        NEWSSCREEN();  
        mvprintw(5,5,"Do you have an ACM program you want to execute?");  
        printw("(y/n) ");  
        refresh();  
        ch = tolower(getch());  
        refresh();  
        while (ch != 'y' && ch != 'n')  
        {  
            mvprintw(10,5,"You must respond with a 'y' or and 'n'");  
            mvprintw(5,5,"Do you have an ACM program you want to execute?");  
            mvprintw("(y/n) ");  
            refresh();  
            ch = tolower(getch());  
            refresh();  
        }  
    }  
    if (ch == 'y')  
        RUN0;  
    else  
        eop = TRUE;  
} /* CHKQUIT */
```

SCREEN MANAGING ROUTINES

This section contains all the routines needed to display, handle and manage the screen and windows.

```
INTRO()
{
    initscr();
    nonl();
    cbreak();
    noecho();
    refresh();
    mvprintw(2,29,"A CONCURRENCY METHOD:");
    mvprintw(4,23,"An Implementation On A 3B2 Network");
    mvprintw(6,40,"by");
    mvprintw(8,34,"JOHN E. MORRELL");
    mvprintw(11,25,"Based On The Model Created By");
    mvprintw(14,29,"DR. ELIZABETH A. UNGER");
    mvprintw(15,28,"Kansas State University");
    mvprintw(22,23,"(C) Copyright 1986 John E. Morrell");
    mvprintw(23,26,"and Kansas State University");
    refresh();
    flash();
    TIMER(3);
} /* INTRO */

QUITSCREEN()
/* Called by MAIN. Screen is displayed after user logs out of ACM */
{
    NEWSCREEN();

    endwin();
} /* QUIT SCREEN */

SHOWREQ(scr,t)
/* Called by SHOWJOBS to display a single request from any of the */
/* lists - Current Request, Ready List */
WINDOW *scr;
struct tt *t;
{
    struct ms *m;

    if (t != NULL)
    {
```

```

if (t->t_stim != NULL)
{
    fprintf(resultfile, "[ %s %s %d ]", t->t_stim->c_name,
        t->t_stim->c_cond, t->t_stim->c_value);
    wprintw(scr, "[ %s %s %d ]", t->t_stim->c_name, t->t_stim->c_cond,
        t->t_stim->c_value);
}
fprintf(resultfile, "%s(", t->t_name);
wprintw(scr, "%s(", t->t_name);
m = t->t_results;
while (m != NULL)
{
    if (m->m_rcvd)
    {
        fprintf(resultfile, "%d", m->m_value);
        wprintw(scr, "%d", m->m_value);
    }
    else
    {
        fprintf(resultfile, "%s", m->m_name);
        wprintw(scr, "%s", m->m_name);
    }
    m = m->m_next;
    if (m == NULL)
    {
        fprintf(resultfile, ";");
        wprintw(scr, ";");
    }
    else
    {
        fprintf(resultfile, ",");
        wprintw(scr, ",");
    }
}
m = t->t_materials;
while (m != NULL)
{
    if (m->m_rcvd)
    {
        fprintf(resultfile, "%d", m->m_value);
        wprintw(scr, "%d", m->m_value);
    }
    else
    {
        fprintf(resultfile, "%s", m->m_name);
        wprintw(scr, "%s", m->m_name);
    }
    m = m->m_next;
    if (m == NULL)
    {
        fprintf(resultfile, ")");
    }
}

```

```

        wprintw(scr,"");
    }
    else
    {
        fprintf(resultfile, "");
        wprintw(scr,"");
    }
}
if (t->t_term != NULL)
{
    fprintf(resultfile, "[ %s %s %d ]", t->t_term->c_name,
        t->t_term->c_cond, t->t_term->c_value);
    wprintw(scr, "[ %s %s %d ]", t->t_term->c_name, t->t_term->c_cod
        t->t_term->c_value);
}
}
fprintf(resultfile,"0");
} /* SHOWREQ */

```

TRAVERSE(p, lines)

/* Called by SHOWJOBS to display jobs on the Blocked List. */
 struct tt *p;

```

{
    int i,j;

    for (i = MINREQ; i <= MAXREQ; ++i)
        if (p->t_next[i] != NULL)
        {
            if (lines < 9 && p->t_next[i]->t_onreadylist == FALSE)
            {
                j = 0;
                mvwprintw(blkwin,j+1,0,"%s", p->t_next[i]->t_uname);
                do
                {
                    waddch(blkwin,p->t_next[i]->t_path[j]);
                    ++j;
                }
                while (p->t_next[i]->t_path[j] != 0);
                waddstr(blkwin,"");
                SHOWREQ(blkwin, p->t_next[i]);
                ++lines;
            }
            else
            {
                if (lines > 8)
                    return(TRUE);
                TRAVERSE(p->t_next[i], lines);
            }
        }
} /* TRAVERSE */

```

```

SHOWJOBS()
{
    struct tt *r;
    int lines;
    int j;
    int more;

    /* Display Current Request in Execution at this terminal */
    wmove(curwin,1,0);
    wclrbot(curwin);
    fprintf(resultfile, "SCREEN NUMBER: %d0, screennum);
    if (now != NULL)
    {
        j = 0;
        mvwprintw(curwin,1,0,"%s",now->t_uneame);
        do
        {
            waddch(curwin,now->t_path[j]);
            ++j;
        }
        while (now->t_path[j] != 0);
        waddstr(curwin, " ");
        SHOWREQ(curwin, now);
    }
    else
        mvprintw(curwin,1,0,"NO JOB IS CURRENTLY IN EXECUTION");
    wrefresh(curwin);

    /* Display Requests in Ready List */
    wmove(readywin,1,0);
    wclrbot(readywin);
    if (ready.front != NULL)
    {
        r = ready.front;
        lines = 0;
        while (r != NULL && lines < 9)
        {
            j = 0;
            mvwprintw(readywin,j+1,0,"%s",r->t_uneame);
            do
            {
                waddch(readywin,r->t_path[j]);
                ++j;
            }
            while (r->t_path[j] != 0);
            waddstr(readywin, " ");
            SHOWREQ(readywin,r);
            ++lines;
            r = r->t_next1;
        }
        if (r != NULL)

```

```

        mvwprintw(readywin,9,5,"----- more -----");
        wrefresh(readywin);
    }

    /* Display Requests in Blocked List */
    wmove(blkwin,1,0);
    wclrtoebot(blkwin);
    if (blocked != NULL)
    {
        lines = 0;
        more = FALSE;
        more = TRAVERSE(blocked);
        if (more)
            mvwprintw(blkwin,9,5,"----- more -----");
        wrefresh(blkwin);
    }
    ++screennum;
} /* SHOWJOBS */

```

TREE MANAGER ROUTINES

This section contains all the routines needed to build, parse, and maintain the request-tree structure.

```
struct tt *NODE(level, p, path)
    int level;
    struct tt *p;
    int path[MAXPATH];
{
    int i;
    struct tt *q;

    q = p;
    for (i = 0; i <= level; ++i)
        if (q != NULL)
            q = q->t_next[path[i]];
    return(q);
} /* NODE */
```

```
WAITO
{
    if (hold.front == NULL)
    {
        hold.front = t;
        hold.rear = t;
        t->t_next1 = NULL;
    }
    else
    {
        hold.rear->t_next1 = t;
        hold.rear = t;
        t->t_next1 = NULL;
    }
} /* WAIT */
```

```
WHATFILE(fname)
    char fname[S15];
{
    char ch;
    int i;

    NEWSCREEN();
    noecho();
    mvprintw(5,5,"Enter the PROGRAM file name: ");
```

```

refresh();
i = 0;
ch = getch();
while (ch != 015)
{
    addch(ch);
    fname[i] = ch;
    i++;
    refresh();
    ch = getch();
}
} /* WHATFILE */

```

```

CONDENSE(infile, tempfile)
{
    int ch;

    ch = fgetc(infile);
    while (ch != EOF)
    {
        if (ch != 040)
            fputc(toupper(ch), tempfile);
        ch = fgetc(infile);
    }
} /* CONDENSE */

```

```

int ISITNUM(m)
struct ms *m;
{
    char    temp[S80];
    int i;
    int start;
    int finished;

    start = 0;
    finished = FALSE;
    if (m->m_name[0] >= '0' && m->m_name[0] <= '9')
        i = 1;
    else
        if (m->m_name[0] == '.' && m->m_name[0] >= '0' &&
            m->m_name[1] <= '9')
        {
            i = 2;
            start = strlen(m->m_name);
        }
    else
        i = 0;
}

```

```

if (i == 0)
    m->m_rcvd = FALSE;
else
{
    m->m_value = (float)strtod(m->m_name);
    if (m->m_value != -1)
    {
        m->m_rcvd = TRUE;
        strcpy(m->m_name,"");
        if (i == 2)
            m->m_value = -(m->m_value);
    }
    else
        finished = TRUE;
}
return(finished);
} /* ISITNUM */

```

ADDQUAD(t)

```

{
    /* Add the statements needed for execution of the Quadroot function */
    /* 1. Q1(AC,A2,NEGB,BB;A,B,C,2.0) */
    /* 11.MULTIPLY(AC;A,C) */
    /* 12.MULTIPLY(A2;A,2.0) */
    /* 13.SUB(NEGB;0.0,B) */
    /* 14.SQUARE(BB;B) */
    /* 2. Q2(AC4;AC,4.0) */
    /* 21.MULTIPLY(AC4;AC,4.0) */
    /* 3. Q3(RADSR;BB,AC4) */
    /* 31.SUB(RADSR;BB,AC4) */
    /* 4. Q4(RAD;RADSR) */
    /* 41.SQRT(RAD;RADSR) */
    /* 5. Q5(R11,R12;NEGB,RAD) */
    /* 51.SUB(R11;NEGB,RAD) */
    /* 52.ADD(R12;NEGB,RAD) */
    /* 6. Q6(R1,R2;R11,R12,A2) */
    /* 61.DIVIDE(R1;R11,A2) */
    /* 62.DIVIDE(R2;R12,A2) */
} /* ADDQUAD */

```

int PARSE(data, start)

```

char *data;
int start;
{
    int ok;
    int finished;
    int i,j;
    int size;

```

```

char    input[80];
char    ch;
struct ms *m;
char    temp[80];

size = strlen(data);

/* The following line causes a bus error and core dump */
/* The program works fine to here. */
strcpy(&t->t_name, user);
t->t_next1 = (struct tt *) malloc(sizeof(struct tt));
t->t_complete = FALSE;
t->t_onreadylist = FALSE;
ctermid(t->t_station);
t->t_done = FALSE;
ok = TRUE;

/* Copy the data into the input array */

i = 0;
while(!isspace(*data))
    input[i++] = *data++;

/* Check for STIMULATION condition */

if (input[start] == '[')
{
    start = 1;
    t->t_stim = (struct cn *) malloc(sizeof(struct cn));
    i = start;
    while (input[i] != '>' || input[i] != '<' ||
           input[i] != '=')
        ++i;
    strcpy(t->t_stim->c_name, input, i-start-1);
    start = i-1;
    i = start;

    while ((input[i] == '>' || input[i] == '<' ||
           input[i] == '=') && i <= 2)
        ++i;
    strcpy(t->t_stim->c_cond, input, i-start-1);
    start = i-1;
    i = start;

    while (input[i] != ']')
        ++i;
    strcpy(temp, input, i-1);
    t->t_stim->c_value = (float)strtod(temp);
    start = i;
}

```

```

t->t_stim->c_val = 0.0;
t->t_stim->c_rcvd = FALSE;
}

/* Check for the NAME of the request */
while (input[i] != '(')
    ++i;
if (i > start)
{
    strcpy(t->t_name, input, i-start-1);
    start = i;
}
else
    ok = FALSE;

t->t_leaf = TRUE;
if (ok)
    if (t->t_name == "ADD")
        t->t_time = 1;
    else if (t->t_name == "SUB")
        t->t_time = 1;
    else if (t->t_name == "ADDMATS")
        t->t_time = 1;
    else if (t->t_name == "MULTIPLY")
        t->t_time = 2;
    else if (t->t_name == "DIVIDE")
        t->t_time = 2;
    else if (t->t_name == "POWER")
        t->t_time = 2;
    else if (t->t_name == "SQRT")
        t->t_time = 2;
    else if (t->t_name == "SQUARE")
        t->t_time = 2;
    else if (t->t_name == "SINE")
        t->t_time = 5;
    else if (t->t_name == "COS")
        t->t_time = 5;
    else if (t->t_name == "TAN")
        t->t_time = 5;
    else if (t->t_name == "NL")
        t->t_time = 5;
    else if (t->t_name == "EXP")
        t->t_time = 5;
    else if (t->t_name == "TROOT")
        t->t_time = 5;
    else if (t->t_name == "QUADROOT")
    {
        t->t_time = 5;
        t->t_leaf = FALSE;
    }
/* ADDQUAD(t); */
/* The above routine will be added later */

```

```

    }
    else
        t->t_leaf = FALSE;
        t->t_time = 0;

/* Check the struct ms for the request */
if (ok)
{
    finished = FALSE;
    t->t_results = (struct ms *) malloc(sizeof(struct ms));
    m = t->t_results;
    while (!finished)
    {
        i = start;
        j = start;
        while (input[i] != ',') ++i;
        while (input[j] != ',') ++j;

        if (i != start)
            if (j != start)
                if (i < j)
                {
                    strcpy(m->m_name, input, i-start-1);
                    finished = ISITNUM(m);
                    ok = finished;
                    start = i;
                    m->m_next = (struct ms *) malloc(sizeof(struct ms ));
                    m = m->m_next;
                }
                else
                {
                    finished = TRUE;
                    strcpy(m->m_name, input, j-start-1);
                    finished = ISITNUM(m);
                    ok = finished;
                    start = j;
                    m->m_next = NULL;
                }
            else
            {
                ok = FALSE;
                finished = TRUE;
            }
        else
            if (j == 0)
            {
                ok = FALSE;
                finished = TRUE;
            }
            else
            {

```

```

        finished = TRUE;
        strcpy(m->m_name, input, j-start-1);
        start = j;
        finished = ISITNUM(m);
        ok = finished;
        m->m_next = NULL;
    }
} /* while */

if (ok)
{
    finished = FALSE;
    t->t_materials = (struct ms *) malloc (sizeof(struct ms));
    m = t->t_materials;
    while (!finished)
    {
        i = start;
        j = start;
        while (input[i] != ',') ++i;
        while (input[j] != ',') ++j;

        if (i != start)
        {
            strcpy(m->m_name, input, i-start-1);
            finished = ISITNUM(m);
            ok = finished;
            start = i;
            m->m_next = (struct ms *) malloc (sizeof(struct ms));
            m = m->m_next;
        }
        else
        {
            if (j != start)
            {
                finished = TRUE;
                strcpy(m->m_name, input, j-start-1);
                finished = ISITNUM(m);
                ok = finished;
                if (m->m_name == NULL)
                    m->m_rcvd = TRUE;
                start = j;
                m->m_next = NULL;
            }
            else
            {
                ok = FALSE;
                finished = TRUE;
            }
        }
    } /* while not finished */
} /* if ok */
}

```

```

if (input[start] == '[')
{
    ++start;
    t->t_term = (struct cn *) malloc (sizeof(struct cn));
    i = start;
    while (input[i] != '>' || input[i] != '<' ||
           input[i] != '=')
        ++i;
    strcpy(t->t_term->c_name, input, i-start-1);
    start = i-1;
    i = start;

    while ((input[i] == '>' || input[i] == '<' ||
           input[i] == '=') && i <= 2)
        ++i;
    strcpy(t->t_term->c_cond, input, i-start-1);
    start = i-1;

    i = start;
    while (input[i] != ']')
        ++i;
    strcpy(temp, input, i-start-1);
    t->t_term->c_value = (float)strtod(temp);
    start = i;
    t->t_term->c_val = 0.0;
    t->t_term->c_rcvd = FALSE;
}

if (ok)
{
    i = start;
    while (isspace(input[i])) ++i;

    if (i != start)
        strcpy(t->t_comments, input, strlen(input));
    else
        strcpy(t->t_comments, "    ");
}

if (ok)
    for (i = MINREQ; i = MAXREQ; ++i)
        t->t_next[i] = NULL;

if (ok)
    return(TRUE);
else
    return(FALSE);
} /* PARSE */

```

```

PUTINTREE(p)
struct tt *p;
{
    int i,j;
    struct tt *q;

    j = 0;
    do
    {
        q = p;
        p = p->t_next[t->t_path[j]];
        ++j;
    }
    while (j <= MAXREQ && t->t_path[j] != 0);
    q->t_next[t->t_path[j-1]] = t;
    t->t_parent = q;
} /* PUTINTREE */

```

```

ADD_TO_TREE()
{
    int i, j;
    struct tt *q;
    struct ms *m;

    j = 0;
    for (i = 0; i <= MAXPATH; ++i)
        if (t->t_path[i] != 0)
            ++j;
    if (j == 1)
        PUTINTREE(blocked);
    else
    {
        if (t->t_leaf)
        {
            q = NODE(j-2, blocked, t->t_path);
            if (q != NULL)
                PUTINTREE(blocked);
            else
                WAIT();
        }
        else
        {
            m = t->t_materials;
            t->t_complete = TRUE;
            while (m != NULL)
            {
                if (!m->m_rcvd)
                    t->t_complete = FALSE;
                m = m->m_next;
            }
        }
    }
}

```

```

        if (t->t_complete)
            PUTINTREE(blocked);
        else
            WAITO;
    }
}
} /* ADD_TO_TREE */

```

BUILDTREE(infile,outfile,resultfile)

```

{
    char    *input;
    struct tt *t;
    int     i, j, k, start;
    int     done, notfull;
    i = MINREQ;
    done = FALSE;
    while (!done)
        if (i <= MAXREQ)
            if (blocked->t_next[i] == NULL)
            {
                done = TRUE;
                notfull = TRUE;
            }
            else
                ++i;
        else
        {
            notfull = FALSE;
            done = TRUE;
        }
    level = 0;
    if (i <= MAXREQ)
        path[level] = i;
    else
        path[level] = 0;
}

```

fprintf(resultfile,"ORIGINAL STATEMENTS FROM INPUT FILE0);

TIMER(3);

while (fgets(input,80,infile) != NULL)

```

{
    j = 0;
    start = 0;
    t = (struct tt *) malloc(sizeof(struct tt));
    fprintf(resultfile,input);
    TIMER(3);
    for (i = 0; i <= MAXPATH; ++i)
        t->t_path[i] = 0;
    for (i = 0; i <= level; ++i)
        t->t_path[i] = path[i];
    while (toupper(input[j]) != '.')

```

```

    ++j;
    k = i;
    while (k < i + j - 1)
    {
        t->t_path[k] = atol(input[start]);
        ++k;
        ++start;
    }
    ++start;
    PARSE(input,start);
    if (ok && notfull)
        ADD_TO_TREE();
    else
        WAIT();
}
} /* BUILDTREE */

```

```

LOADFILE()
{
    char ch;

    WHATFILE(fname);
    strcpy(res,fname);
    strcpy(out,fname);

    strcat(fname, ".acm");
    infile = fopen(fname, "r");

    if (infile == NULL)
    {
        mvprintw(10,1,"Cannot open %s for reading.", fname);
        mvprintw(11,1,"%s does not exist.",fname);
        mvprintw(13,1,"Make sure the file has the extension '.acm.'");
        refresh();
    }
    else
    {
        outfile = fopen(strcat(out, ".off"), "w");
        tempfile = fopen("temp.acm", "w");
        resultfile = fopen(strcat(res, ".res"), "w");

        CONDENSE(infile, tempfile);

        fclose(infile);
        fclose(tempfile);
        tempfile = fopen("temp.acm", "r");

        BUILDTREE(tempfile, outfile, resultfile);
    }
}

```

```
        fprintf(resultfile, "Oinished loading %s into the tree.", fname);  
        fclose(tempfile);  
    }  
} /* LOADFILE */
```

A CONCURRENCY METHOD:
AN IMPLEMENTATION ON A 3B2 NETWORK

by

JOHN E. MORRELL

B.M.E., Fort Hays State University, 1978

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Abstract

More and more tasks in today's society require quick processing of large and difficult operations. Most of the computers in use today are based on the sequential processing techniques used by early electronic computers. The need for methods and machines which allow true concurrent processing is great. Computing machines developed for the concurrency required in future applications should have the capability to execute concurrent programs. Data flow computers are believed to be one way of achieving a high level of concurrency in a sensible, easily programmed manner.

This thesis describes an implementation for "A Concurrency Method", a model based on the principles of data-driven sequencing and the single assignment of data items. The system is designed for and implemented on a network of AT&T 3B2 super-microcomputers, which are based on the AT&T WE-32000 32-bit microprocessor. This implementation provides a central kernel of the operating system which can be made to take on the characteristics of a data flow computer by utilizing the message passing capabilities of AT&T's local area network, 3BNet, to link all of the processors on the network. A specialized operating system incorporating 3BNet and the processing power of the 3B2/300 super-microcomputers is needed to provide a simulation of a naturally concurrent processing system.

This implementation of ACM includes an operating system with a compiler for executing ACM statements, screen management routines for the display of of program execution, and a tree-structure management system. The system includes the basic arithmetic and trigonometric primitive operations for mathematical computations. No editor was implemented for this

system since the UNIX operating system, on which the implementation runs, provides three editing packages.

The system was designed with three goals in mind: (1) to provide a working simulation of a data flow computer based on the concepts of ACM; (2) to allow a better understanding of concurrent processing, as related to distributed systems and parallel processing computers; and (3) to help in the visualization of other research areas involving the need of concurrent computation by allowing other students to implement various communication schemes around this central kernel.